Contributions to software deployment in a component-based reflexive architecture

Jakub Kornaś

UNIVERSITÉ JOSEPH FOURIER

No attribué par la bibliothèque

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UJF

Specialite : Informatique : Systèmes et Communication

préparée au laboratoire LIG, projet SARDES,
dans le cadre de l'Ecole Doctorale
Mathématiques, Sciences et Technologies de l'Information

préparée et soutenue publiquement par

Jakub KORNAŚ

le 23 Octobre 2008

Contributions au déploiement dans les architectures réflexives basé sur les composants

---

Directeur de thèse :

Jean-Bernard STEFANI

JURY

M.	Pierre	SENS	Université Paris 6	Président du Jury
M^{me}.	Françoise	BAUDE	Université de Nice	Rapporteur
M.	Lionel	SEINTURIER	Université Lille 1	Rapporteur
M.	Thierry	COUPAYE	Orange Labs	Examinateur
M.	Olivier	GRUBER	Université Joseph Fourier	Examinateur
M.	Jean-Bernard	STEFANI	INRIA Rhône-Alpes	Directeur de thèse

Acknowledgments

First and foremost, I would like to thank my family – Jadwiga, Stanisław and Gosia – for always being there for me and supporting me in the difficult moments. Without you three this document would have probably never existed. I would also like to thank Izabela, who has been my best friend for so many years now. And Agata, with whom I hope to spend the rest of my life and who reminded me what the important things are.

I would like to thank Jean-Bernard Stefani for taking me on this PhD journey and being the advisor of this thesis.

Olivier Gruber for his tremendous help in finalizing this work and for all the interesting discussions that we had.

Adrian Mos for being my best friend here in Grenoble. I have learnt a great deal of stuff from you, man! And it ain't all work-related :)

Julien Legrand, Noël de Palma, Matthieu Leclercq, Fabienne Boyer, Benoït Claudel, Stéphane Fontaine, Jérémy Philippe, Sylvain Sicard and Christophe Taton, my INRIA colleagues, with whom I have enjoyed working as well as simply spending time. Hope to see you in Kraków soon!

And finally I would like to thank all the members of the SARDES project for the time spent together.

Résumé de thèse

Les logiciels récents sont de plus en plus complexes en terme de leur développement et gestions associés. Pour adresser cette complexité, un approche fondé sur les composant a vu le jour qui vise une meilleure ingénierie des logiciels. En particulier, une approche à composants structure les logiciels comme un assemblage dynamique des composants qui doivent être déployés et gérés à l'exécution, en continu. Pour ce deploiement et cette gestion, nous avons adopté une approche fondée sur une architecture logicielle explicite de composants. Cette approche, communément appelée "gestion basée sur l'architecture", a évolué de premières solutions ad-hoc vers des infrastructures génériques fondées sur des modèles de composants réflexifs. Une de ces infrastructures est la plateforme JADE.

JADE vise à gérer autonomiquement des systèmes distribués complexes. Basé sur un modèle de composants réflexifs, JADE capture l'architecture logicielle complète des systèmes distribués, incluant non seulement les applications distribuées hébergées mais aussi les systèmes distribués qui les "hébergent". En particulier, cette architecture réifie en continue certains aspects de l'exécution des systèmes distribués, tels que les défaillances de noeuds ou les caractéristiques de performance. Utilisant cette architecture réifiée, les gestionnaires autonomes observent et réagissent selon les changements de conditions. Chaque réaction de gestionnaires automatiques a pour but de planifier une reconfiguration de l'architecture en réponse à un changement des conditions d'exécution. Par exemple, un gestionnaire d'auto-réparation observant la défaillance d'un noeud aurait pour but de reconstruire, sur un autre noeud, la partie perdue du système distribué. Un gestionnaire d'auto-protection observerait une intrusion et planifierait la reconfiguration du système distribué pour isoler les composants compromis. Un gestionnaire d'auto-optimisation pourrait observer une disponibilité en baisse d'un serveur répliqué et planifier daugmenter cette réplication cardinale des composants serveurs.

Au coeur de cette gestion autonomique fournie par JADE se trouve le déploiement des composants. En effet, la plupart des reconfigurations de l'architecture de systèmes distribués s'appuient sur l'aptitude à instancier des composants sur des noeuds distants. Plus précisément, une fois que les gestionnaires autonomes ont générés un plan de reconfiguration du système distribué, l'exécution effective du plan est automatiquement distribuée, essentiellement par création et suppression de composants ainsi que leur édition de liens. La création et la suppression de composants requièrent une gestion locale des composants sur chaque noeud du système distribué. Cette gestion locale nécessite une infrastructure distribuée pour trouver, installer, charger et supprimer les composants.

Le travail présenté dans cette thèse est le socle de JADE, fournissant les capacités de déploiement avancé dans un environnement distribué. En particulier, nous avons traité ce défi via une conception récursive o l'implémentation de composants a été modélisée par des composants. Cette approche fournie un déploiement uniforme qui suit les principes basés sur l'architecture. En particulier, nous pouvons déployer non seulement des composants d'applications mais aussi des composants "middleware". De plus, au-delà du déploiement de composants normaux, nous pouvons également déployer des logiciels "legacy" qui sont gérés uniformément par JADE.

En plus de fournir le socle du déploiement de JADE, ce travail a montré que le modèle de com-

posant réflexif utilisé par JADE (appelé FRACTAL) a besoin d'être étendu pour capter les implémentations et leur spécificité. Bien que conu spécifiquement pour FRACTAL et JADE, ces extensions présetent une applicabilité bien plus large, elles s'appliquent en effet à la plupart des modèles de composant courant. En effet, la plupart de ces modèles se focalisent sur lassemblage fonctionnel de composants et ne proposent rien ou peu sur le déploiement des composants. Ce travail a également montré que l'architecture autonomique basée sur l'architecture a des besoins dynamiques spécifiques en terme de déploiement qui rend difficile la ré-utilisation de plateformes existantes pour la gestion dynamique de composants, tels que OSGI. Sur la base d'OSGI, cette thèse a expérimenté plusieurs conceptions et prototypes qui ont été utilisé avec succès, mais dont ultimement les limites sont apparues. Ces limites sont liées à une tension fondamentale entre les conceptions architecturales de JADE et OSGI. Une nouvelle conception et un nouveau prototype ont montré la faisabilité de supporter notre FRACTAL étendu sur la plateforme Java, servant de fondation à la gestion autonomique de systèmes distribués complexes fondée sur larchitecture.

Contents

0.4pt0.3pt 0pt0.0pt 0pt0.0pt 0pt0pt

List of Figures

Résumé de chaptire 1

Dans le chapitre 1 de cette these on introduit nos travaux de recherche dans la domaine de deploiement des logiciels a base des composants. D'abord on explique les problemes de deploiement des logiciels dans le contexte des systems autonomes base sur l'architecture explicite des logiciels. Ensuite, on donne une definition precise de ce que le deploiement des logiciels dans ce contexte est. Puis on explique les contributions scientifiques de cette these.

Chapter 1

Introduction

Contents

1.1 Challenges of component-based software deployment

Modern software systems are increasingly complex, both regarding their development and their runtime management. To address these challenges, a component-based paradigm has emerged. Components not only modularize development for better software engineering but also promote a vision of software as a dynamic assembly of components that can be deployed and managed at runtime. One successful approach for advanced runtime management is based on the software architecture that components make explicit. This approach, called architecture-based management, has evolved from ad-hoc early solutions to generic frameworks based on reflexive component models. One such framework is the JADE framework.

JADE targets autonomic management of complex distributed systems. Based on a reflexive component model, JADE captures the complete software architecture of distributed systems, including hosted distributed applications and their hosting distributed system. In particular, this architecture reifies the runtime behavior of the distributed system, such as node failures or performance characteristics. Using this complete architecture, autonomic managers can then observe and react accordingly to changing conditions. The reaction of autonomic managers is to plan an architecture reconfiguration. For instance, a self-repair manager observing a node failure would plan to reconstruct the lost part of the distributed system on another node. A self-protection manager would observe an intrusion and would plan reconfiguring the distributed system in order to isolate or sandbox compromised components. A self-optimizing manager could observe an availability problem in a replicated server and plan to augment the replication cardinality of the server components.

Consequently, at the heart of JADE, one finds the challenge of component deployment. Indeed, most reconfigurations of the architecture of a distributed system rely on the ability to instantiate components on remote nodes of the distributed system. More precisely, once autonomic managers have generated a reconfiguration plan of the distributed system, the actual execution of the plan is itself distributed, essentially creating and removing components as well as binding and unbinding them.

Creating and removing components requires a local management of components on each node of the distributed system. This local management requires a distributed infrastructure to find, download, install, and remove components.

The work presented in this thesis has provided JADE with advanced deployment capabilities in a distributed environment. In particular, we have approached this challenge through a recursive design where the implementation of components has been modelled using components, providing a sound and uniform deployment that follows the architecture-based principles. In particular, we can deploy application components as well as middleware components. Moreover, we can not only deploy regular components but we can also deploy legacy software that are wrapped with components. Through wrapping, remotely deployed legacy software can be managed by JADE in a uniform manner.

Besides providing the deployment subsystem of JADE, this work has shown that underlying component model, a reflexive component model called FRACTAL, needed to be extended in order to capture implementations and their specifics. Although conceived for FRACTAL and JADE, these extensions apply to most current component models that focus on the functional assembly of components and not on how components are deployed. This work has also shown that autonomic architecture-based management has specific dynamic needs in terms of deployment that makes it difficult to reuse existing dynamic component platforms such as OSGI. A first design and prototype, although successful in many ways, has demonstrated fundamental design and architectural tensions between OSGI and JADE. A second design and prototype has shown the feasibility of using the Java platform for supporting our extended FRACTAL as the foundation to autonomic architecture-based management of complex distributed systems.

In the following sections, we first put deployment in perspective with respect to the overall challenge of software management. We then present the contributions of this work in more details. We conclude this introduction with presenting and discussing the organization of this document.

1.2 What software deployment is?

Deployment of a software component spans multiple phases, as represented in figure 1.1 (taken from (Hall 2004)). It covers all the activities performed on a given component after it has been developed by the software developer. Once the component has been developed, it can enter the *release* phase of deployment. A released component can then be *installed* on the target nodes on the consumer side. Installed components can then be managed, namely *activated* and *deactivated*, *updated*, *reconfigured* and *removed* if they become obsolete.

The specific roles of deployment phases are as follows:

- The **release** phase is performed right after the component has been developed and compiled against a set of dependencies. The result of this phase is a packaged software component ready to be installed on target machines. A component package contains information on the dependencies that the given component needs in order to work (unless the component package is self contained). Furthermore, a package may contain different implementations of the interfaces exposed by the given component. Normally, a component is released into a repository, for example on a web server, from which it can be *installed* on the target machines.

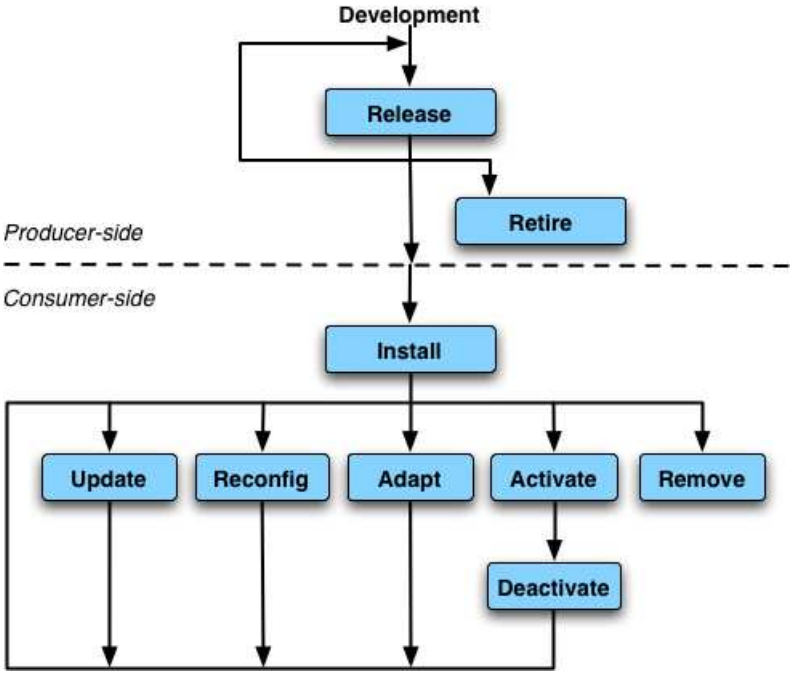


Figure 1.1: The deployment life cycle of a software component

- The **install** phase consists in "physically" bringing the component onto the target machine where it will execute. Once the component is installed, it can be run on that machine.

- The **activate** and **deactivate** phases correspond to the launching and stopping down the component

- The **update** and **adapt** phases are similar. An update consists in replacing a component with a newer version which has been released. In most systems this means installing a new version of an already present component on the target machine, deactivating the old version and activating the new one. In systems that support *dynamic* updates the deactivation is not necessary and as a result the state of an updated component is not lost. The adapt phase modifies the configuration of an installed component and requires it to be restarted.

- The **reconfigure** consists in changing the configuration parameters of an installed component. This deployment phase is similar to the adapt phase, however reconfigurations can be performed on running components.

- The **remove** step of deployment means that a given component is undeployed from the target machine. Usually this phase of deployment is performed after the component has been deactivated. The removal of a component results in freeing the underlying resources, such as processing power, disk space etc.

- The **retire** phase is performed on the repository where a component has been released. When a component is retired, it can not be installed any more on the targets.

Based on the analysis of related work which we perform in the first part of this thesis we conclude that most of the existing research projects around component deployment and architecture-based management, such as SmartFrog (Goldsack et al. 2003) and DAnCE (Deng et al. 2005), as well as existing distributed middleware frameworks, such as CORBA (Group 1996), RMI (Sun Microsystems Inc. 1996) and JMS (*Java Message Service Specification Final Release 1.1* 2002), do not address the issue of component installation, versioning and dynamic updates. These projects simply assume that implementations of components are readily available on target nodes. On the other hand, deployment solutions addressing the non component-based software applications, such as the DEB and RPM projects as well as the .NET framework can not be used in the architecture-based management context because they do not provide a component model on top of which the explicit reflexive architectures and their autonomic managers could be built.

Therefore, in this thesis we focus on the post-release phases of component deployment in the context of autonomic, architecture-based management, as we believe those aspects of component deployment to be insufficiently addressed by existing solutions. Our work environment – the JADE architecture-based autonomic management system – in its initial version only handled the *activate* and *deactivate* as well as *adapt* and *reconfigure* phases of component deployment. It did not handle any of the component deployment phases which require the manipulation of the components' implementations, such as *installation*, *updates* etc.

1.3 Contributions of this thesis

In our first attempt to enhance JADE with post-release deployment functionality, we have applied OSGI as the deployment layer. OSGI, as will be described in detail in the first part of this thesis, is currently the most advanced deployment solution for Java applications. It provides a packaging format, a runtime environment for isolation and versioning of Java code and a set of tools for the development, publishing, and on-target installation of pieces of this code. Thus, using OSGI as a deployment layer for JADE seemed highly interesting.

Based on OSGI, we have implemented a system for distributed deployment of Java components which provides several interesting characteristics compared to the state of the art. Firstly, it supports architecture-based deployment in which functional assemblies of components in terms of provided and required services are described together with the implementations of these components. We call it the architecture-based deployment, because in this approach the deployment related information, such as target nodes and physical packages, becomes an integral part of the software architecture. This information can be provided in, for example, an Architecture Description Language (ADL) file. Secondly, the OSGI-based deployment system supports component versioning and, in certain cases, dynamic updates. Finally, integrating OSGI and JADE has many advantages in terms of reuse of the state of the art development and software publishing tools for Java provided by OSGI and Eclipse. This implementation of the JADE platform proved successful in terms of real world usability and is now part of the JASMINE project. It supports and simplifies deployment of various industrial middleware.

Unfortunately, the OSGI-based implementation of the deployment system for JADE turned out to be incompatible with the architectural approach to autonomic software management. The principal

issue with using OSGI as a mechanism of component installation and versioning is that it hides several important aspects of the reflexive architecture within its runtime—the decision on how components are resolved in terms of Java types is taken by the OSGI runtime and not exposed by it to JADE managers. This contradicts the JADE approach to software management, where JADE managers take decisions on architectural modifications.

Therefore, the main rationale behind our "post OSGI" work was to answer the following question: how do we extend JADE and its component model in order to provide equivalent deployment capabilities but without OSGI? Although this work is based on and was integrated in the JADE platform, we believe the approach and core concepts to be applicable to most of the existing architecture-based management frameworks for software components such as SmartFrog or Rainbow (Cheng et al. 2004). From the high-level point of view, the deployment-related extensions to component-based reflexive architectures essentially mean capturing dependencies between component implementations by introducing the notions of *modules* and *physical packages* into the component model.

A module is about the loaded implementation of a component. The application components, such as the ones initially managed by JADE, are about a service-oriented architecture, in which dependencies between components are those of provided and required services. If a component model is implemented in an object-oriented programming language, this means a resolution of dependencies based on service objects which implement interfaces. Modules are also components, thus they also have dependencies which need to be resolved. The dependencies between modules represent implementation dependencies. For example, if a component model is implemented in the Java programming language, a dependency between module components is about the classes in one module imported by classes in another module. If the model is implemented in C or C++, the inter-module dependency is about unresolved symbols, such as functions and global variables.

As regular components need to have their dependencies resolved, modules also need their dependencies resolved. This resolution has however the semantics of code-level linking rather than service-level binding. The mechanisms used are however exactly the same and relies on FRACTAL components and bindings. For components implemented in Java, the implementations rely on Java class loaders and module resolution is about controlling class loading delegation between these class loaders. For components implemented in C or C++, resolution is essentially about the linking of code, often packaged as dynamically loaded libraries (DLLs). One important point in both cases is that the assembly is dynamic, so resolved modules may be unresolved, which requires reorganizing class loader delegation or unlinking DLLs, taking into consideration the limitations of each execution engine regarding such reconfigurations at runtime.

A physical package is about a physical container that encapsulates the on-disk implementation of one or more components. The notion of a physical package, similarly like the one of modules, strongly depends on the programming language in which a component model is implemented. For example, a physical package would correspond to a JAR archive if the component model is implemented in Java and to a file on the file system or an ELF archive if the model is implemented in C or C++. Physical packages require a notion of *local repository*. This repository is where the packages are stored on a given node. From there, they can be associated to module components. The local store can have, like for example in NIX, a complex hierarchical structure managed similarly to what is done in terms of memory management. Another abstraction associated with the physical packages is the one of

garbage collection—unused physical packages are cleaned-up automatically.

Another important extension to the component model is the one we call the *resolver*. Resolver is the entity that binds (connects) components to form a reflexive architecture. It therefore resolves the dependencies between components. Since we introduce modules as special types of components, as a consequence we also distinguish two types of inter-component resolution—the resolution of traditional applicative components and the resolution of module components. By contrast to the standard version of JADE, in which component implementations were not considered, in the extended version, the resolver of module components is part of the explicit architecture of the JADE system.

In addition to the extensions to a component model, we also propose a layered approach to the deployment of component-based software. This approach distinguishes a local (single machine) deployment layer and the distributed (spanning many machines) layer. A local layer is about the management, isolation, and versioning of implementations of components, whereas the distributed layer is about the synchronized execution of the deployment process (plan) on multiple target machines. We believe that the problems encountered in the local and distributed deployment layers are sufficiently disparate to be handled by two distinct abstractions. This allows for a more structured and technology-independent approach to modelling deployment.

We believe our approach to be the first attempt at building an integrated environment for component development, deployment and runtime management. We have validated it by building a prototype full-featured architecture-based management system for JADE, without the drawbacks of the OSGI-based solution.

1.4 Thesis overview

This thesis is divided into three parts: the analysis of the state of art, the description of the contribution and finally the evaluation of our work.

Part I – Analysis of the state of art The goal of the first part of the thesis is to investigate the existing work around the deployment of component-based software. Since this is a vast area, we have divided our analysis of the related work into three domains: the *Models and frameworks for the deployment of software components*, the *Software packaging systems* and finally the *Module systems for Java*. Each of those aspects of investigation is treated in a separate chapter. We describe component-based as well as “standard” approaches to these issues.

Deployment frameworks and models for component-based software are described in Chapter 2. Our main goal in presenting the work around this subject is to understand how the existing research and industrial projects handle the issue of component deployment. We are specifically interested in distributed, architecture-based deployment, as we believe it to be the state of the art in software deployment.

In Chapter 3 we focus on the issues around the software packaging. It is the building block of software deployment, yet it is an often overlooked aspect of it. We essentially focus on investigating how the existing solutions handle the issues of package dependencies, as well as whether or not they address the software packaging in the context of software components.

Chapter 4 presents an analysis of the existing solutions around the modularity (code versioning

and isolation) in the Java platform. We have chosen Java because (1) it is a programming language and an execution platform often used to implement component frameworks as well as deployment systems on top of them and (2) the prototypes that we have built for this thesis were implemented in Java. Our main goal in this chapter is to see whether any of the existing solutions to the lack of modularity in Java can be coupled with a component-based approach, reflexive software architectures and the requirements on dynamicity and autonomic management.

From those three chapters we build a general vision of the various aspects of software deployment. We also identify a number of limitations of the existing solutions which gives us a foundation for discussing the contributions of our work. We summarize our analysis of related work in Chapter 5.

Part II – Contribution Chapter 6 presents the contribution of this thesis – capturing deployment as an integral part of architecture-based software management. We describe a set of extensions to a component model which bring deployment from the status of external and necessary evil to the status of an autonomic manager, fully integrated with the rest of the architecture-based approach to software management.

Even though we make reference to the FRACTAL component model and the JADE autonomic system, we believe our approach to be independent of the concrete component technology as well as programming language used to implement it.

Part III – Implementation Chapters 7 and 8 describe the experimental evaluations of our approach to component-based software deployment.

Chapter 7 describes the OSGI-based platform for deployment of component-based Java software. This platform was developed as part of this thesis. We show how the approach and the model presented in Chapter 6 are implemented to provide a full-featured deployment platform for Java components. It also shows how this deployment system has been applied to real-world component-based software. As use-cases, we show how various Java middleware, including the Joram JMS server and the JOnAS Java EE server, are deployed by the system.

After identifying certain limitations of the OSGI-based approach, we present in Chapter 8 the second prototype that we have built during this thesis. This prototype does not use OSGI. Instead, it directly implements the extensions to a component model and builds a deployment solution from scratch. Even though less complete in terms of tooling and practical applicability than the solution described in Chapter 7, this prototype has the advantage of supporting the full gamut of dynamic deployment capabilities needed by autonomic architecture-based management.

Conclusion Chapter 9 concludes this thesis and outlines the areas of future investigation.

Part I

Analysis of the state of art

Résumé de chaptire 2

Dans le chapitre 2 de cette thèse on présente les modelés, les API et les plateformes pour le déploiement et la reconfiguration dynamique des logiciels a base des composants. D'abord on décrit la spécification OMG D&C, qui est indépendante de technologie subjacent. Ensuite on présent la plateforme de déploiement et configuration DAnCE, qui est une implémentation de la spécification OMG D&C et qui vise les systèmes embarques de temps réel. On continue en décrivant le JSR88 — une spécification pour le déploiement des composant Java Enterprise Edition . Enfin on illustre plusieurs solutions existants pour le déploiement et la configuration des compostants logiciels, tel que le SmartFrog, le Fractal Deployment Framework (FDF) et autres. Notre but dans ce chapitre est de expliquer comment les problèmes de déploiement des composants logiciels sont abordé par l'état de l'art et de identifier les dsavantages des solutions existants.

Chapter 2

Models and frameworks for the deployment of component-based software

Contents

This chapter presents existing models, APIs and frameworks for the deployment and dynamic reconfiguration of component based software systems. We first describe the OMG Deployment and Configuration (D&C) specification, which is independent from the underlying component technology. Next we discuss the DAnCE deployment and configuration framework, which is an implementation of the OMG D&C specification and targets the Distributed Real-time and Embedded (DRE) systems. We continue by describing JSR88, a deployment Application Programming Interface (API) specification for Java Enterprise Edition (Java EE) component-based applications. Finally, we illustrate several existing deployment and configuration frameworks such as SmartFrog, Fractal Deployment Framework (FDF) and others. Our goal is to investigate how the subject of deployment of software components is addressed by the state of the art, as well as to identify the shortcomings of existing solutions.

2.1 OMG D&C

The Object Management Group (OMG) has defined a specification called Deployment and Configuration (D&C) of distributed component based applications (Group 2006). The goal of this specification is to unify and thus facilitate the deployment and configuration of component based systems into a set of heterogeneous, distributed target nodes. The specification achieves this unification by defining models that describe the deployment process in a platform-independent manner. The OMG

D&C follows a Model Driven Architecture (MDA) approach—it defines several Platform Independent Models (PIMs) which are transformed into Platform Specific Models (PSMs) for concrete component based technologies. The OMG D&C specification also standardizes several aspects of deployment and configuration of component based distributed systems by precisely defining deployment-related abstractions and identifying the interactions between these abstractions.

An application, as defined by this specification, is a component that is assumed to be independently useful. It can either be implemented directly, as a monolithic implementation, or as an assembly of subcomponents, where each subcomponent can again, recursively, be either monolithic or an assembly. Ultimately, each application can be decomposed into a set of monolithic components. At deployment time decision is made about which components are to be deployed where.

A component package, according to the OMG D&C specification, is a set of metadata and compiled code containing implementations of a component interface. Packages can reference other packages. This property allows for reusing third-party implementations as well as replacing dependent packages without the impact on the rest of the application. However, no on-line update functionality is implied by the OMG D&C specification—this feature is considered optional by the specification. Component packages can contain multiple implementations to provide a match for a given target environment, such as Windows, Linux etc.

2.1.1 Deployment process

The deployment process starts once the software has been developed, packaged according to the D&C specification and published by a software provider. It is then acquired by a new owner, called the *deployer*. The deployment process consists of the *installation*, *configuration*, *planning*, *preparation* and *launch* phases. The *installation* phase means moving the published software package into the repository. The location of the repository is not necessarily the same as the location where the software will execute. Once the software is in the repository it can be configured, but only in terms of functional configuration and not in terms of where it will execute. The decision on how and where software components are deployed is taken in the *planning* stage. This phase also consists in deciding which implementation of the software component will be deployed. The result of this stage is the *deployment plan*, specific for a given target environment. The *preparation* phase means performing the work needed to deploy a given component on a given target node. This phase can consist in moving the binary files from the repository into the target machines. The preparation can either be done “just in time” or performed in advance in order to minimize the start-up time. The *launch* phase consists in bringing the application into the executing state. This means interconnecting and configuring component instances as well as starting execution. Application runs until it completes or until it is stopped by the same infrastructure that launched it.

2.1.2 Deployment models

Following the phases of the deployment process, the D&C PIMs can be divided into three main models: the *Component Model*, the *Target Model* and the *Execution Model*. The component model describes component assemblies in terms of bindings, encapsulation etc. The role of the target model is to represent the environment into which the components are deployed. Finally, the execution models

represent the runtime aspects of component assemblies.

From another point of view, the PIMs can be divided into the *Data Models* and the *Management Models*. Data models are used to generate and store descriptive information about component assemblies, configurations and deployment constraints. The management models specify runtime entities (managers) that create, process, provide and store the data model content in order to deploy a system.

In total six different models can be identified within the OMG D&C specification:

- The Component Data and Management model
- The Target Data and Management model
- The Execution Data and Management model

Below we describe those six different models and their role in representing the component deployment process.

Component Data and Management Model

Each component deployable in conformance with this specification has interfaces, via which it can be connected to other components, and has an implementation, which is either monolithic or composite. The component data model provides abstractions that describe these components, their implementations and interfaces as well as information on component packages and configuration. A simplified view of the component data model is presented in figure 2.1.

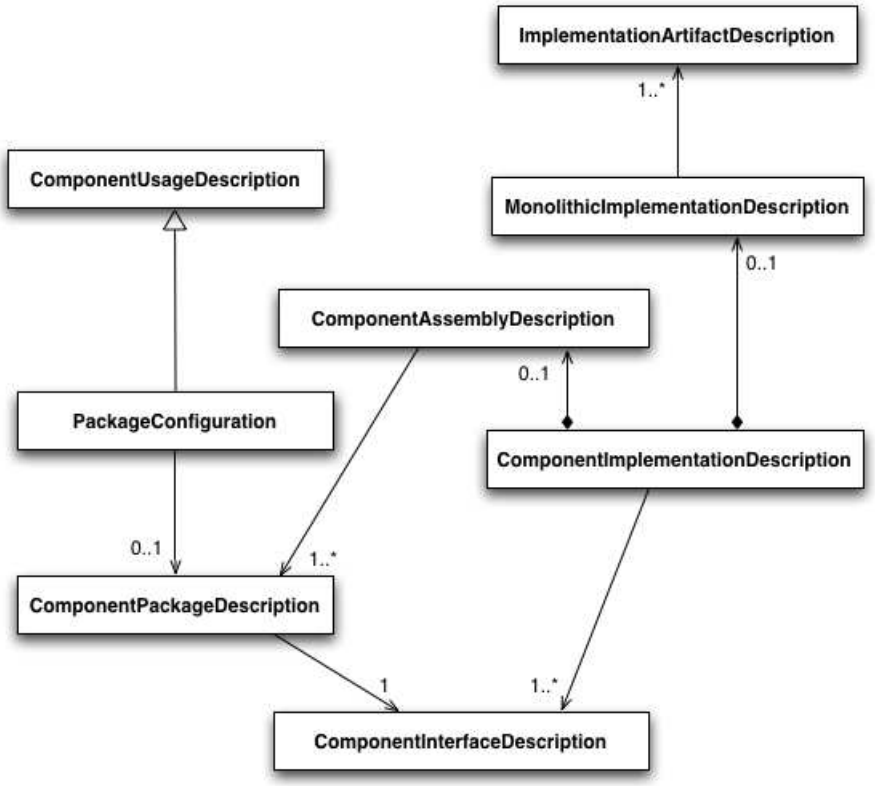


Figure 2.1: Data part of the OMG D&C component model

The *PackageConfiguration* describes one configuration of a component package and represents a reusable work product. It extends the *ComponentUsageDescription*. This allows for several ways of specifying the *ComponentPackageDescription* that is being configured. The *ComponentPackageDescription* describes alternative implementations of the same component interface. It references the *ComponentInterfaceDescription* and the *ComponentImplementationDescription* and is composed of one or more *PackagedComponentImplementations*.

The component management model only contains one entity: the *RepositoryManager*. This manager exposes methods such as *installPackage*, *createPackage*, *findPackageByName* and *deletePackage*. It manages component data by maintaining a collection of *PackageConfiguration* elements.

Target Data and Management Model

The target model describes and manages information about the environment in which applications are deployed.

The target data model defines abstractions that represent the target environment. A simplified UML diagram of this model is represented in figure 2.2. The top-level abstraction is the *Domain*, it contains *Nodes*, *Interconnects*, *Bridges*, *Resources* and *SharedResources*. The *Domain* simply wraps information about its content. The *Node* has computational capabilities and is a target for executing component instances. *Nodes* can either use *Resources* or *SharedResouces*. *Interconnects* and *Bridges* can only use *Resources*—they can not share them. *Interconnects* provide direct connections between

Nodes whereas the *Bridges* provide routing capabilities between the *Interconnects*.

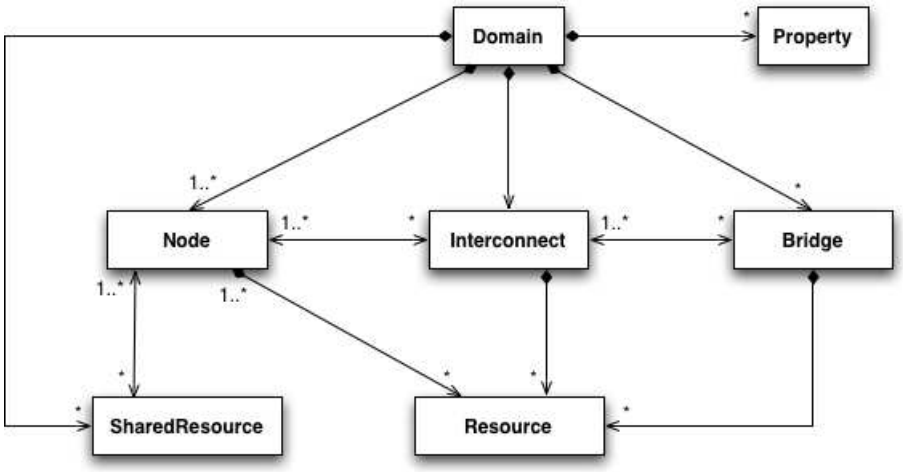


Figure 2.2: Data part of the OMG D&C target model

The target management model consists of only two management abstractions—the *TargetManager* and the *ResourceCommitmentManager*. The *TargetManager* provides information about the domain in a form of its data model, as well as tracks the usage of resources within this domain. It provides methods to create and destroy the *ResourceCommitmentManagers*. The *ResourceCommitmentManagers* allow for reservation/dereservation of resources.

Execution Data and Management Model

The main abstraction in the data part of the execution model is the *DeploymentPlan*. The deployment plan is the result of the *planning* stage of the deployment process and can either be executed immediately or stored for further use. It contains information about: (1) artefacts that are part of the deployment via the *ArtifactDeploymentDescription* abstraction, (2) how to create component instances from these artifacts (the *MonolithicDeploymentDescription*) and (3) where to instantiate the components (the *InstanceDeploymentDescription*).

The management part of the execution model contains the following abstractions: the *ExecutionManager* which manages the deployment process for all domains. It interacts with a set of *DomainApplicationManagers*. A *DomainApplicationManager* is responsible for the deployment of components within a single domain. It splits the deployment plan for a given domain into several subplans—one for each node. The *NodeManager* runs on each node. It manages the deployment of components on a given node, without taking under consideration the application to which these components belong. Components are created by containers. Each container is hosted in a process called the *NodeApplication*. The *NodeManager* uses the *NodeApplicationManager* to create the *NodeApplication* process. *NodeApplicationManager* is responsible for deployment of components within a *NodeApplication*. One *NodeApplicationManager* is created for each deployment in order to differentiate deployments in a node. A *NodeApplication* provisions resources (CPU, memory, etc.) for the components that it hosts. It creates an environment for instantiation of components based on the metadata from the deployment plan, which it obtains from other managers.

A simplified diagram of the management part of the execution model is presented in figure 2.3.

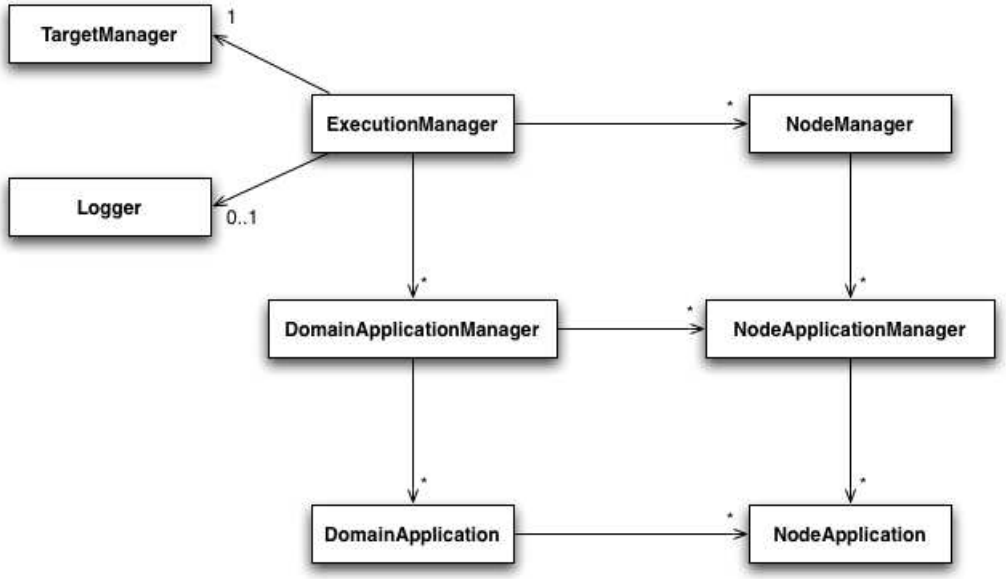


Figure 2.3: Management part of the OMG D&C execution model

RepositoryManager is associated to a domain and is used by the deployer agents to store component implementations and by *NodeApplicationManagers* to retrieve the software packages. Components are stored as dynamic linking libraries and are transferred by the *NodeApplicationManager* to the local node's file system when needed.

2.1.3 Summary

The OMG D&C specification is very interesting for several reasons. Firstly, to our knowledge it is the only existing attempt at fully modelling the deployment process. The OMG D&C platform independent models cover not only the component-based application itself, but also the environment into which this application is deployed as well as the execution of the deployment process and the execution of the deployed application. Furthermore, this specification is independent from the underlying component technology, which makes it highly reusable. Finally, it contains a highly interesting notion of the deployment plan. The specification proved its real-world applicability in projects such as DAnCE, which we describe in the following section of this chapter.

Despite the above advantages, the OMG D&C specification also has certain drawbacks. Firstly, its platform specific models (PSMs) support only a single component technology. Namely, it is impossible to deploy with this specification an application built with two different component technologies. This has been identified and addressed in (Hnetynka 2005). Secondly, the modelling of component implementations in this specification is limited. Nothing is said about the resolution of dependencies between implementations of software components, as well as on how this resolution integrates into the deployment process. As will be described in this thesis, it is an important issue of component deployment.

2.2 DAnCE

DAnCE (Subramonian et al. 2007) is a deployment and configuration engine aimed at distributed real-time and embedded component based systems (DRE). It conforms to the OMG D&C specification and thus can be seen as an implementation of it. Authors of the DAnCE project argue that following a component based approach to software development, therefore separating the application functionality from lifecycle activities, brings important advantages in terms of software flexibility. Such flexibility is highly important in software systems such as shipboard computing environments, inventory tracking systems and intelligence, surveillance and reconnaissance systems, which are the target application domains of the DAnCE engine.

Authors of DAnCE consider the conventional component platforms, such as Java EE and .NET, not well-suited for real time or embedded systems, because they do not provide real-time quality of service (QoS) support. Moreover, applying the component based approach to the construction of DRE middleware introduces many challenges that hinder its adoption. These challenges are essentially the need to:

- Deploy the component assemblies into the target nodes
- Activate and deactivate component assemblies automatically
- Initialize and configure server resources to enforce end-to-end QoS requirements

According to the authors, there are no portable, reusable and standard mechanisms to address these challenges. Therefore, DAnCE provides the following capabilities:

- Automatic downloading of component packages
- Automatic configuration of Object Request Brokers (ORBs), containers and component servers
- Automatic establishment of bindings between components
- Automatic deployment and lifecycle management of middleware services (such as security, transactions etc.)

In its current implementation, DAnCE uses XML files as deployment descriptors and deploys only CCM components. Since DAnCE follows the OMG D&C specification, its deployment mechanism implements and works with the abstractions that we have described in Section 2.1. DAnCE defines *domains* as the target execution environments. Domains consist of *nodes*, *interconnects*, *bridges* and *resources*. System to be deployed by dance is described in a *deployment plan*. Architecture of the DAnCE deployment system is presented in figure [fig]. All the entities in DAnCE are implemented as cooperating CORBA services. *ExecutionManager* manages the deployment process for all domains and interacts with a set of *DomainApplicationManagers*, which in turn are responsible for the deployment of components within a single domain. *DomainApplicationManagers* split the deployment plan for a given domain into several sub plans – one for each node – and pass the extracted information to the *NodeManagers* running on the nodes. *NodeManagers* manage the deployment of components on a given node, without taking into consideration the application to which these components belong.

Components are created by containers. Each container is hosted in a process called the *NodeApplication*. The *NodeManager* uses the *NodeApplicationManager* to create the *NodeApplication* process.

NodeApplicationManager is responsible for deployment of components within a *NodeApplication*. One *NodeApplicationManager* is created for each deployment in order to differentiate deployments in a node.

NodeApplication provisions resources (CPU, memory, etc.) for the components it hosts. It creates an environment for instantiation of components based on the metadata from the deployment plan, which it obtains from other managers.

By providing the implementation of the *RepositoryManager* service, DAnCE has the functionality to store component implementations and retrieve different versions of them. The *RepositoryManager* can also retrieve the software packages given a URL.

DAnCE *NodeApplicationManagers* can periodically poll their associated *RepositoryManagers* in order to obtain the newest version of a software package. This way, when an administrator decides to redeploy a component, an overhead induced by downloading the new software package from the repository into the target node can be minimized. Software packages in DAnCE are ZIP files transported via CORBA invocations.

DAnCE also defines a standard lifecycle for its software components, consisting of four states: *PREACTIVE*, *ACTIVE*, *PASSIVE* and *DEACTIVATED*. In a *PREACTIVE* state a component has been created and all its environment settings have been performed. Once a component is activated with its underlying middleware, it is in an *ACTIVE* state. *PASSIVE* state indicates that the component is not executing and its required resources can be used by other components in the application. A *DEACTIVATED* component is one which has been removed from the system. DAnCE components can only be connected and activated if they are in the *PREACTIVE* state. Similarly, components can only be deactivated if they are in the *PASSIVE* state.

Furthermore, DAnCE allows the user to specify, using XML files, the end-to-end QoS related deployment information. Namely, one can manipulate the ORB command line-options, such as choosing a communication protocol or optimizing request processing and the ORB service configuration options, such as concurrency and demultiplexing models. This is an important feature required by the real-time software components which DAnCE is able to deploy.

Summary The main goal of the DAnCE project is to provide a QoS-enabled environment for deployment and management of software components. The target application domain of this work are the distributed, real time and embedded systems (DREs). In such systems, assuring the quality of service is important because the systems must often support real-time processing for tasks such as avionics mission computing and shipboard computing. However, despite a precisely defined application domain, DAnCE is as a matter of fact a full-featured deployment platform for software components which can be used in other contexts than DRE systems.

DAnCE addresses the challenges of component deployment by following the OMG D&C specification. As a result, the deployment framework is independent of the underlying component technology. The main contribution of DAnCE is extending the OMG D&C model with QoS-related aspects. From the functional point of view, the framework supports XML-based deployment of components. In the XML-based architecture description file one can specify not only the desired component as-

sembly, but also its QoS requirements. The DAnCE framework will then take care of the distributed deployment of the components and of the proper configuration of middleware services according to the QoS constraints. Finally, the DAnCE platform performs an automatic installation of component binaries.

DAnCE shares many similarities with the work described in this thesis, especially in terms of architecture-based deployment and dynamic installation of their implementations. However, DAnCE does not attempt to model this installation. It also is not based on a reflexive component model, thus offers limited runtime management capacity. Furthermore, as described in (Deng et al. 2005) component implementation binaries are stored in the form of dynamic linking libraries, which has many drawbacks.

2.3 JSR88

JSR88 (*J2EE Deployment Specification (JSR88)* 2005) is a deployment API (Application Programming Interface) for Java Enterprise Edition (Java EE). Java EE is an environment for building and running component-based, distributed, multi-tiered applications. Two types of Java EE components exist: the *WEB* components, which usually provide the presentation tier (static or dynamic web pages) and the *EJB* (or Enterprise Java Beans) components, which provide the business logic of the application. Those two types of components are different not only in terms of functionality, but also packaging - web components packages, or *WARs* (Web ARchives), usually contain static (HTML) and dynamic (JSP) web pages, whereas the *EJB-JAR* archives contain compiled Java classes. All the Java EE components are deployed into a Java EE server.

The goal of JSR88 is to allow tools from different providers to configure and deploy Java EE applications on any Java EE server. JSR88 identifies three roles participating in the process of deploying a component-based application:

- The *Java EE product provider* is a supplier of a Java EE server (container) in which the application components can be deployed
- The *deployment tool provider* is a supplier of the software tool that can be used in development, packaging, deployment, management and monitoring of an application
- The *deployer* is a person or a software programme who configures and deploys a Java EE application component on a specific product (server). Typically this is done in three steps: configuration, distribution and execution.

In order to make the deployment tools independent of the Java EE products and also to allow the deployers to use different deployment tools, the JSR88 specifies an API that has to be implemented by the Java EE product and the deployment tool, as well as a model of the information exchanged between those three participants when deploying a Java EE component.

In order to be compatible with JSR88, the product needs to implement interfaces defined in the *javax.enterprise.deploy.spi* package, namely the *Deployment Manager*, the *Deployment Factories* (for accessing the deployment manager) and the *Deployment Configuration Components* (JavaBeans). The

main role of the product is to generate the platform-specific deployment information, as well as to deploy the application.

The tool, which usually is an Integrated Development Environment (IDE) such as Eclipse or NetBeans, needs to implement the interfaces defined in the *javax.enterprise.deploy.model* package. In addition, it has to be able to discover and interact with Java EE product's *Deployment Manager*. The main role of the tool is to access the Java EE application archive as well as to display and allow for editing of the deployment-related information extracted from the archive. Therefore, the JSR88 interfaces implemented by the deployment tool simply represent the information exchanged with the Java EE product.

From the sequential point of view, the deployment happens as follows:

- The tool opens an application archive file (such as an EJB-JAR or a WAR) and builds a *DeployableObject* object from it. This object represents the metadata of the application archive and is part of the deployment model.
- The tool obtains a reference to the *DeploymentManager* and asks it for the server-specific information for a given *DeploylableObject*, via the *DeploymentConfiguration createConfiguration(DeployableObject)* method.
- The *DeploymentConfiguration* is made editable to the deployer, usually through the graphical user interface (GUI) of the deployment tool.
- Once the deployment information was filled-in by the deployer, he chooses the *Target*s on which the module will be deployed, by calling the *Target[] getTargets()* method of the *DeploymentManager*
- Next, the tool asks the *DeploymentManager* to distribute the configured module on selected targets. This phase consists in the installation of the configured Java EE modules "inside" the containers provided by the Java EE products represented as targets. A module installed within a target is represented by the *TargetModuleID* abstraction.
- The tool can now ask the *DeploymentManager* to perform life cycle operations on the given installed modules

The life cycle operations that one can perform on a deployed Java EE module are *start/stop* but also *redeploy* and *undeploy*. The start and stop operations are fairly straightforward. The redeploy operation is interesting, because its goal is to replace the given module with its updated version. The undeploy operation's goal is to completely remove a given module. Both redeploy and undeploy operations are however optional in the specification, thus are not necessarily implemented by the server products.

Summary Main goal of the JSR88 is to separate the deployment tools from the products on which Java EE applications are deployed. Therefore, the deployers do not need to know all the details of specific deployment tools in order to deploy applications on different Java EE servers and also any

deployment tool can work with any Java EE product. This can be considered an important advantage of JSR88 compared to other deployment APIs.

On the other hand, JSR88 is specific to the Java EE technology and is not generic enough to be used outside this context. Furthermore, even within the context of Java EE components, this specification fails to address several important issues. First of all, since this JSR essentially works with physical packages (EJB-JARs, WARs etc.) it does not fully model the software components that it deploys and the dependencies between them, as identified and described in detail in (Exertier 2004). This is contrary to the definition of a component, which is a fully introspectable software entity.

Furthermore, the details of installation, isolation and dynamic updates of the software components deployed with this JSR are not covered by the specification. As a result, Java EE server product providers apply ad-hoc solutions to provide execution-time isolation and dynamic updates of Java EE components, which will be described in detail in section 4.2. These solutions are usually based on class loaders and often have drawbacks resulting in class name clashes and other unexpected interactions between the application components. Moreover, they usually force the deployer of the application to learn about the internal deployment mechanisms of a given Java EE product.

2.4 SmartFrog

SmartFrog (Goldsack et al. 2003) is a deployment and reconfiguration framework for Java. The main reason behind the existence of this framework, created by HP Labs, is to overcome issues of incorrect configuration in distributed systems, such as the multi-tier web services, where often numerous software components have to be correctly configured, bound and started in a proper sequence. According to the authors, no existing tools ensure the above and as a result, ad-hoc, error-prone solutions are adopted by system administrators. SmartFrog aims to (1) increase operational reliability (resilience to failure) (2) reduce cost of installation and maintenance and (3) assure correctness and consistency. To achieve it, the framework provides a *configuration language*, a *deployment tool* and an *execution environment* for distributed software applications written in the Java programming language and conforming to the SmartFrog API.

Figure 2.4 presents the overall architecture of the SmartFrog system.

Configuration language The SmartFrog configuration language can be considered an Architecture Description Language. Its principal goal is to allow the application deployer to describe the system to be deployed in a relatively simple, domain-specific language.

The essential information contained in the SmartFrog application descriptions describes:

- What components should be deployed and their configuration parameters
- How components are related in terms of connections and encapsulation
- The workflow associated with lifecycle of components and the whole system

SmartFrog descriptions can be hierarchical, they can be parametrized and can be validated before deployment. The validation is performed against a set of rules, such as dependencies between various system components (e.g. version dependencies), rules governing repetition (e.g. each web server

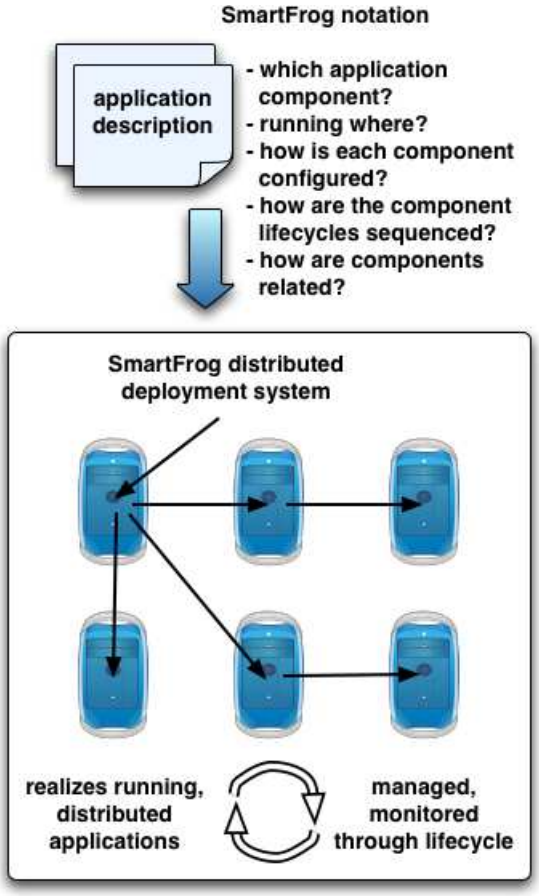


Figure 2.4: SmartFrog architecture

should run the N processes), replication (e.g. two cooperating instances of this component should exist for reliability), location (e.g. this component should be close to the database), and so on. In practice, the validation rules are defined in SmartFrog templates.

SmartFrog descriptions are ordered collections of attributes, where each attribute has a name-value format. However, different types of attributes exist in SmartFrog, namely the *simple values* (such as integers, strings etc.), *references to other attributes* and finally the *references to other components*, for the inheritance purpose. Furthermore, SmartFrog attributes can either be resolved *normally*, that is before deployment, or they are resolved *lazily*—once all the components have been deployed. Figure 2.5 illustrates a set of example SmartFrog descriptions. The *webServerTemplate* and *dbTemplate* descriptions are referenced within the "main" description, which is called *system*:

As can be seen, the component descriptions are hierarchical and each component within the hierarchy defines a name space.

The above illustrated SmartFrog description language could very well be quite different in terms of form—the SmartFrog parser works in fact with a set of open data structures. Thus, different textual representations of the notation can be supported by the parser.

The SmartFrog configuration language is only used to define the configuration parameters of components and the information on the target machines on which these components should be deployed.

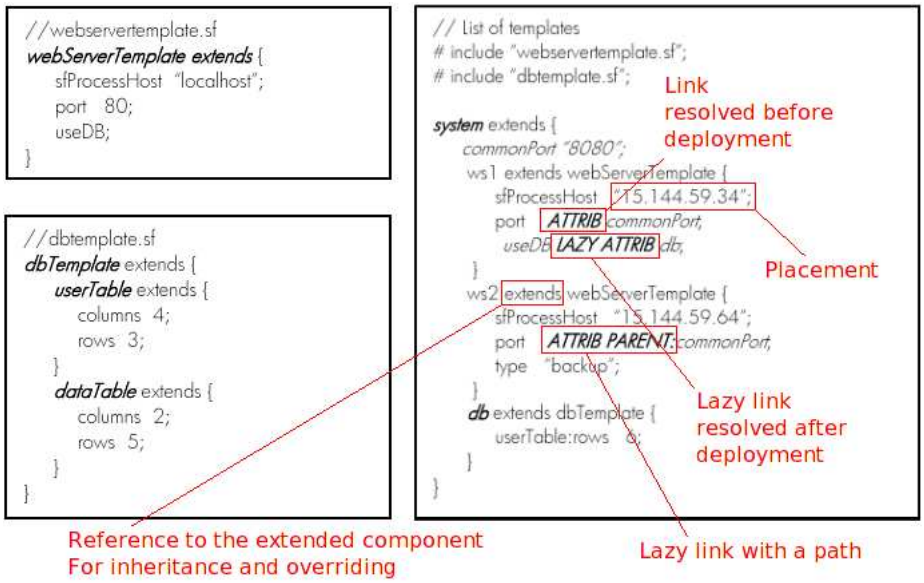


Figure 2.5: SmartFrog description

The notation is not used to define behaviour of components—it is not a programming language. Behavioural part of the component is hidden within the implementation code of these components.

SmartFrog comes with the tools to create, store and manipulate the architecture descriptions written in the SmartFrog configuration language.

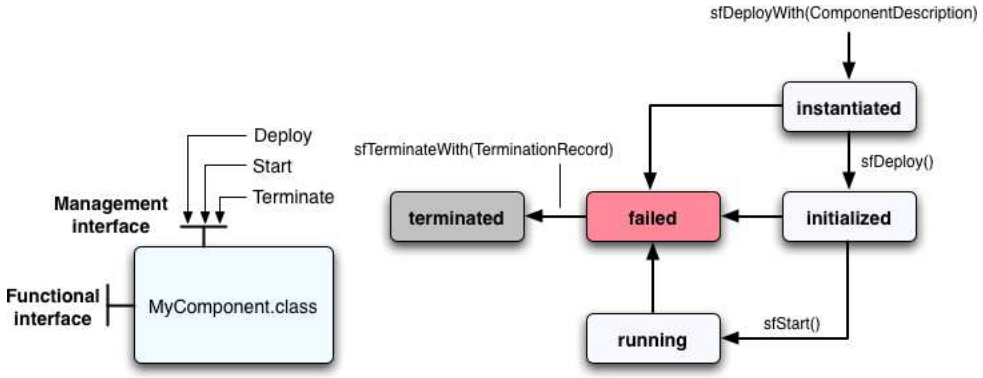
Component model The SmartFrog component model consists of primitive and composite components, it is therefore hierarchical. Furthermore, each SmartFrog component is divided into the control part and the functional part, with the control part being a simple, extensible set of interfaces to access key management actions: instance creation, configuration and termination. SmartFrog components can be:

- Fully integrated, in which case they fully implement the component interfaces directly in Java
- Independent, in which case they are wrappers (Java adapters) for legacy code

SmartFrog distinguishes a notion of a *system* and an *application*. A system deployed and managed by SmartFrog is a collection of applications. Each application is in turn built as a set of components. Since each application consists of a collection of components, an application is not seen by SmartFrog as an atomic object. Each component in an application is tightly bound via a parent-child or binding relationship and the transitive closure of the parent-child relationship defines the scope of the application. There are no direct bindings between components belonging to different applications, but components can locate each other via a naming and discovery service provided by the SmartFrog framework.

Lifecycle of each component in an application is tightly bound to lifecycle of other components via a parent-child relationship - parent components are notified of children death and vice versa. Parents are responsible for the lifecycle of children.

A structure and lifecycle automaton of a SmartFrog component is presented in figure 2.6. The illustrated lifecycle methods, such as the *sfDeployWith(ComponentDescription)* or the *sfStart()* are imposed by the component model and have to be provided by the component's implementation class. Some of these methods, for example the *sfDeployWith(ComponentDescription)* are inherited from the basic component implementation class provided by SmartFrog, and should not be overriden. Other methods, such as the *sfDeploy()* can be overriden to provide component-specific initialization code. All lifecycle methods are called on the component after its implementation class has been instantiated. Some of them make callbacks to component's parent. This is the case for the *sfTerminateWith(TerminationRecord)* method, which notifies the parent component of the given component's termination.



A SmartFrog component SmartFrog components' lifecycle

Figure 2.6: SmartFrog component and its lifecycle

Deployment infrastructure The deployment infrastructure in SmartFrog is a distributed service responsible for the configuration and instantiation of software components. Once the SmartFrog description written in a notation is parsed (processed by the language processor), the simplified model is passed to SmartFrog deployment infrastructure to carry the deployment process. The deployment infrastructure ensures that the deployment tasks are scheduled and executed (orchestrated) in an orderly manner and with transactional guarantees. Once the components are instantiated, the deployment framework can perform initialization operations on them using the "standard", predefined, management interface. Low-level semantics of these management operations can be overloaded by the developer of the application—he can for example define various types of bindings (via a naming service, SLP discovery etc.). Furthermore, SmartFrog provides different possibilities to define the placement of a component—host name, co localization with another component etc.

Physically the deployment means invoking the *sfDeployComponentDescription* method on the ProcessCompound—a predefined SmartFrog composite component which represents a JVM. By default, components from the given deployment description are created in this compounds JVM. This can be changed using *sfProcessHost* parameter. Given a description, compound passes it to a deployer identified by sfProcessHost. Deployer looks for the *sfClass* attribute, that identifies the implementation class of a component, extracts this name and creates an instance of this class using a default

constructor. Component instance is initialized using *sfDeployWith* method. As a result of sfDeployComponentDescription call, an object reference is returned. The receiving object, which usually is the deployer, has to call all the other lifecycle methods in order to start the component.

SmartFrog also comes with a workflow-like system to carry complex configuration tasks on clusters, where task ordering, recovery from failure etc. is required. This system is called the SmartFlow and can be seen as a part of the deployment infrastructure. It consists of (1) a library of composite components with predefined lifecycle (parallel, sequence) etc. to manage the subcomponents, that allow one to build workflows, and (2) of an event framework to disseminate events between components. Currently the SmartFlow lacks transactions and persistence.

Example, predefined SmartFlow components are the following:

- Sequence—Starts components one after another, that is a next component in a sequence is started when the previous one terminates.
- Time-out—Contains a single subcomponent. Starts this subcomponent and waits for it to terminate within a given time.
- Try—Contains several subcomponents. Tries to execute the primary sub-component and when this subcomponent terminates, tries to execute the continuation subcomponent depending on the termination type (the continuation subcomponent is indexed by the termination code).

The deployment infrastructure in SmartFrog also provides interesting solutions in terms of security, based on the public key infrastructure (PKI). Each SmartFrog target machine is a security domain, and is supplied with a security certificate. Furthermore, each software component is signed with a certificate. Based on this, nodes can have different access restrictions and only validated nodes will be allowed, for example, to manipulate components and descriptions that have been correctly signed. Furthermore, all network communication carried out within SmartFrog happens over an encrypted channel.

Summary SmartFrog is an interesting research project around the deployment of highly complex, distributed applications. It has been successfully used in the industrial environment, for example to manage clusters of Java EE servers. It shares many similarities with Fractal Deployment Framework (the FDF), which we describe in the next section.

The interesting aspects of SmartFrog are multi fold. Its component model allows for the modelling of legacy software in a unified manner. The model is hierarchical and divides a component into a control and a functional part. Thanks to the component abstraction, applications deployed and managed by SmartFrog can be described in an architecture description language. In terms of deployment, SmartFrog comes with a sophisticated engine, which is a distributed application. This engine ensures that deployment tasks are executed in an orderly and transactional manner. Furthermore, SmartFrog provides a set of composite components which can be put in the deployment descriptor and which define a specific lifecycle for the children of a given composite (for example sequence, parallel etc.). Finally, the security aspects have been seriously considered and implemented by the creators of SmartFrog.

The main limitation of SmartFrog is that it does not address software packaging, local installation and runtime modularity/versioning issue. It simply assumes that software code is either present on the target machine. Software packages and versions of components instantiated from them are not part of the explicit architectures managed by SmartFrog. Furthermore, SmartFrog does not provide a package repository for its software components. Finally, SmartFrog supports only limited forms of dynamic reconfiguration, dependent on a fixed component life cycle.

2.5 Fractal Deployment Framework (FDF)

Fractal Deployment Framework (FDF), also called DeployWare (*Fractal Deployment Framework* 2006), is a component-based tool for deploying legacy applications. The framework essentially consists of three elements: (1) a language in which the deployer can describe the desired software architecture and (2) an extensible library of Java classes which represent the legacy software that FDF can deploy (such as Apache (*The Apache Software Foundation* 2005) HTTP server, JVMs etc.) as well as the basic deployment operations (SSH, SCP etc.) and (3) a set of tools which facilitate the writing and editing of deployment descriptor files and executing the deployment process. In FDF everything is reified as a component (variables, shells, protocols, commands, software etc.).

Figure 2.7 presents the general architecture of the Fractal Deployment Framework.

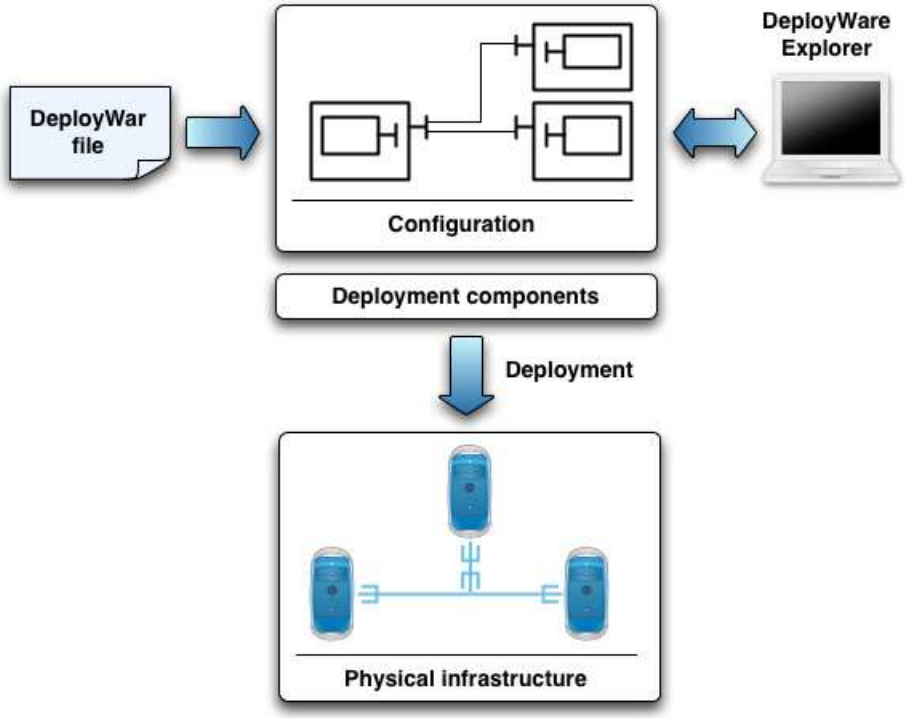


Figure 2.7: General architecture of the Fractal Deployment Framework (FDF)

Given a description of an architecture to deploy (the DeployWare file), FDF first parses it and builds a component-based representation of it, called the configuration. The configuration references the set of predefined FDF deployment components which perform the basic deployment tasks. This

representation can then be edited and manipulated through the tools provided with FDF—the tools allow the end-user to perform the *install*, *configure*, *start*, *stop*, *unconfigure* and *uninstall* operations on the given configuration.

The target software architecture to be deployed using FDF is described in the deployment descriptor, which is either XML-based or is written in a domain-specific language defined by the FDF. The two languages are equivalent in terms of the information they can handle. This high-level description contains the list of components to be deployed and the target host. In the same file, the end-user can reference software components predefined by the FDF, which perform basic deployment actions by wrapping file transfer protocols (FTP, SCP), remote access mechanisms (Telnet, SSH) and different shells (Linux, Windows).

Figure 2.8 is an example of the FDF deployment description for the Petals component.

```
<component name="petals1" definition="PEtALS.SERVER">
    <component name="archive"   definition="PEtALS.ARCHIVE(fileName)"/>
    <component name="home"      definition="PEtALS.HOME(homeDir)"/>
    <component name="name"      definition="PEtALS.NAME(PEtALS-Name)"/>
    <component name="host"      definition="./host1"/>
    <component name="port"      definition="PEtALS.PORT(port)"/>
    <component name="userName"  definition="PEtALS.USERNAME(user)"/>
    <component name="password"  definition="PEtALS.PASSWORD(pass)"/>
</component>
```

Figure 2.8: An FDF description of the Petals component

The FDF configuration language allows the software deployer (the end-user) to define dependencies between components, it also supports certain non-functional properties of deployment, such as parallelism. These dependencies and non-functional aspects of deployment will be then respected by the FDF deployment engine. Thanks to this, one can deploy a complete system in one step—for example, in case of a Java EE application, FDF is capable of deploying everything, starting from the JVM, through the Java EE container, and finally the Java EE application.

Summary FDF can be considered a practical tool for deploying legacy software binaries on heterogeneous machines. Its main advantage, and at the same time similarity with the SmartFrog platform, is that it represents everything as software components. However, since FDF is based on Fractal, its component model is more advanced than the one in SmartFrog—it is not only hierarchical, but also supports sharing of components and is highly extensible.

The main disadvantage of FDF is that it does not address the fundamental issue of packaging and versioning of component-based software. For example, versioning of the library of software components which represent objects deployable with FDF is not supported. In the context of Java applications, FDF does not reify Java types as part of software architecture, thus it is unable to support the evolution of such application without restarting a JVM in which they execute.

2.6 NIX

Nix (Dolstra et al. 2004) is a research project around the deployment of component-based software conducted at the University of Utrecht. The main goal of the project is to provide a framework for *safe* and *flexible* deployment of software components.

By safe deployment the authors of Nix understand the fact that all component's dependencies are satisfied. By flexible they mean that different deployment policies can be defined and that several versions of components can coexist. Nix draws an analogy between deployment and memory management, by identifying two essential problems occurring in deployment:

- Unresolved dependencies, which happen when a component's dependency cannot be found. This is equivalent to a dangling pointer in the file system
- Collisions, which occur when two different versions of a component cannot coexist because they occupy the same location in the file system

Nix proposes an approach to this problem which organizes the component repository like an address space. In this approach, every component has a unique name which is similar to a symbolic name in a file system. In the repository this unique name corresponds to a folder in which all the component's resources are stored. The symbolic name also contains, as a prefix, the hash of all the symbolic names of components used to build a given one. Thus, if a change to any of the "parent" components occurs, a new symbolic name for the given component is generated. This new symbolic name corresponds to a different version of the given component, stored in a new store path.

The closure of component's dependencies in Nix is calculated automatically, based on the principle that if a component A depends on a component B, then A contains the store path of B. This is illustrated in figure 2.9 (taken from (Dolstra et al. 2004)). This figure also illustrates value of the closure.

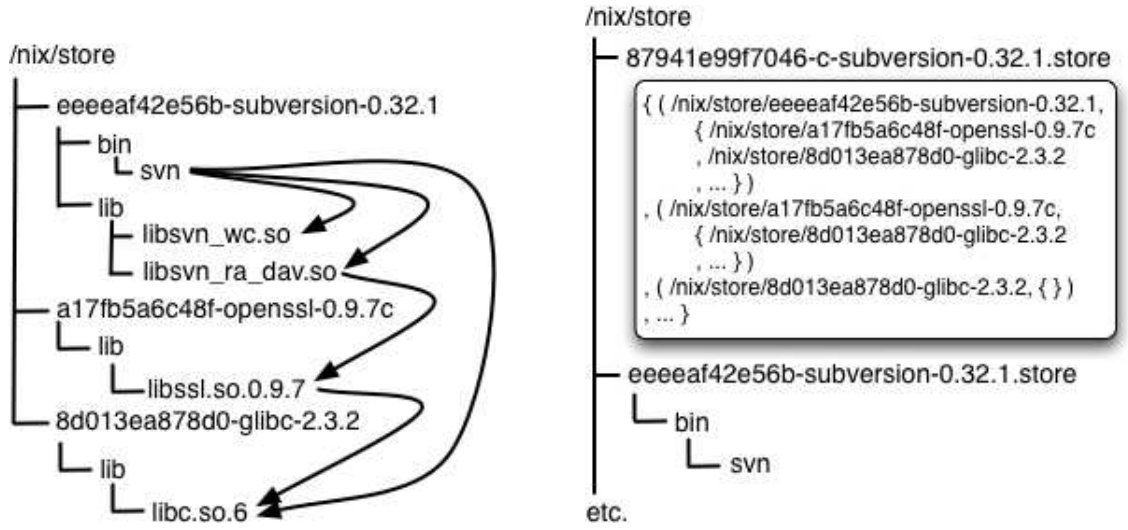
Closures are automatically determined by interpreting derivation values. A sample derivation value is shown in figure 2.10.

A derivation value includes the deployment description (names of components, description of the target platform) and the shell script with the deployment tasks. Normally derivation values are not hand-written, but rather generated from a higher level description. Furthermore, derivation values are hidden from NIX clients using various mechanisms.

The deployment process in NIX is a functional one in that the result of it depends only on the parameters—if we use the same parameters twice, we always obtain the same result. Also, the content of a repository built from a set of inputs never evolves once it has been built.

Summary NIX is a deployment framework which ensures safe and flexible deployment of software components—it solves the problem of unresolved component dependencies and allows multiple versions of components to coexist. Main contributions of NIX are that it manages the component repository like memory and that deployment process in NIX is fully functional one.

However, NIX is strongly tied to the UNIX environment. Furthermore, its expression language is based on shell scripts, thus it is not typed. Finally, NIX was not used for the deployment in a distributed environment so far.



Organisation of the NIX store

A sample NIX closure value

Figure 2.9: NIX repositories and closure values

2.7 Summary

In this chapter we have analysed the existing projects around the deployment of component-based software. We have focused on APIs, models and frameworks for the distributed deployment of such software.

Most of the presented solutions, including SmartFrog, do not address the issue of installing component implementations, the versioning of these implementations and the depencendies between them—they simply assume that software packages are always available on the target machines. The DAnCE system does perform the dynamic installation of implementations of components, but as opposed to SmartFrog and similarly to FDF it does not provide a reflexive component model for runtime management of the deployed architecture. This makes building autonomic systems on top of these three platforms impossible.

NIX and JSR88 on the other hand are technology-specific solutions. NIX was designed with Unix/Linux environment in mind, JSR88 is an API for Java EE applications. Especially the JSR88 is not generic enough to be applicable in a different context than the deployment Java EE components, such as Enterprise Beans and Web Applications.

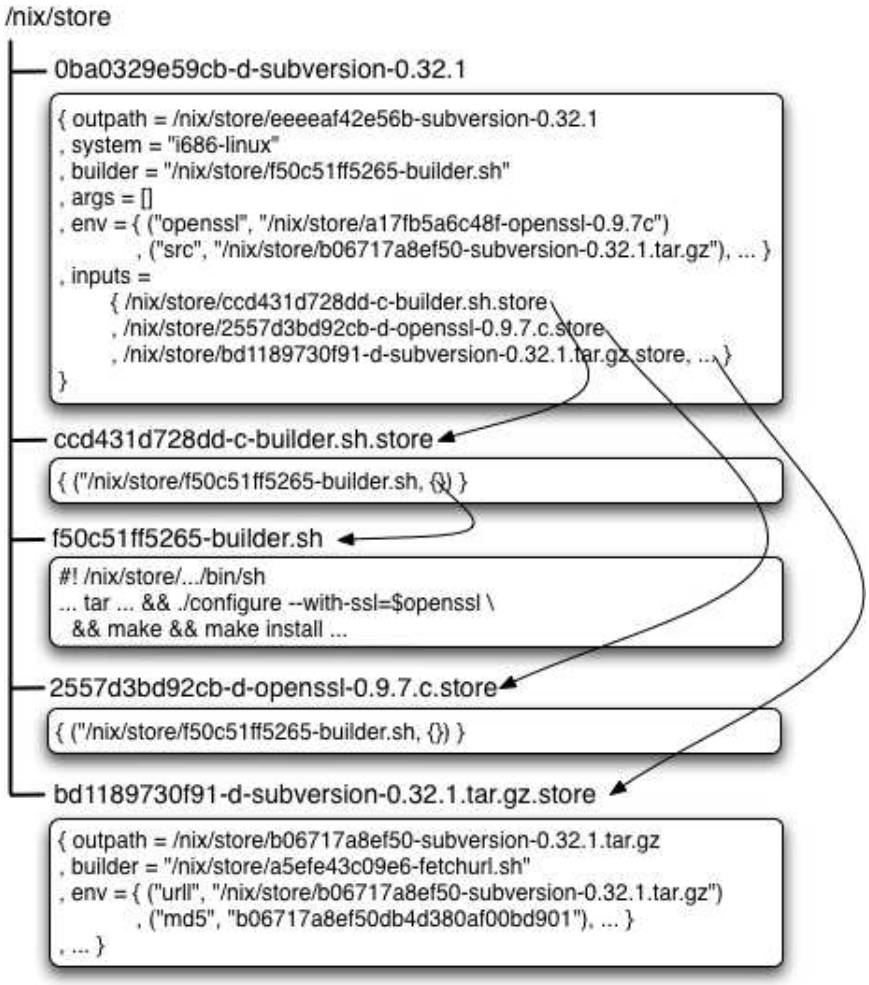


Figure 2.10: Nix derivation value

Résumé de chaptire 3

Dans le chapitre 3 de cette thèse on analyse les systèmes de paquetage des logiciels. On se focalise sur les solutions le plus pertinents—les systèmes de paquetage pour Linux/Unix, tel que les RPMs et les DEBs ainsi que les systèmes de paquetage pour les environnements Java et .NET.

La raison pour nos travaux d'investigation dans cette domaine est le fait que les paquetages physiques et leurs systèmes de gestion constituent une couche de base de déploiement des logiciels. Notamment, pour construire un système de déploiement complet on doit répondre au question d'où le code exécutable des composants arrive, comment il est installé sur les machines cibles et quelle gendre des propriétés les paquetages physiques doivent-elles supporter. On s'intéresse particulièrement aux ensembles des méta-donnes supporté par chaque format de paquetages, tel que les déclarations

des dépendances des pacquages, leur conflits etc.

Chapter 3

Software packaging systems

Contents

This chapter is an anlysis of existing software packaging systems. We focus on the most relevant solutions—the Unix/Linux package formats (DEB and RPM) and the packaging formats for the Java and .NET environments.

The reason for our investigation in this area is that physical packages are the "nuts and bolts" of software deployment. Namely, for a full-featured deployment system one must answer the question of where the executable code of components comes from, how it is installed on the target machines and what kind of properties must the physical packages support. We are especially interested in the set of meta information supported by each of those packaging formats, such as the declarations of package dependencies, conflicts etc.

3.1 Unix/Linux packages

Unix-like operating systems support several types of software packages. Most of those packages can exist in two forms: source packages contain the code which can be compiled on the user's computer, whereas binary packages contain executable, compiled code which only needs to be installed on the user's machine. In this section we will describe two popular package formats for Linux: the DEB and RPM, focusing on the binary packages.

3.1.1 DEB

DEB is a package management system used by the Dpkg—the package manager for the Debian Linux distribution. DEB packages are *archiver* (also known as *ar*) files containing three parts: the package version and two archives containing the package metadata as well as the files to be installed (the package content). Metadata can contain the following fields with information about the package:

- *Package* (mandatory) - a package name

- *Source* - the name of the source package from which this binary package was created
- *Version* (mandatory) - the package version number
- *Section* - a field used to classify packages (*main*, *contrib* or *non-free*)
- *Priority* - a package can be *required* (a system will not function without it), *important* (expected on any Linux system, but the system can work without it), *standard* (installed by default), *optional* (usually installed but not necessary), *extra* (everything else). Packages on the optional level and above need not conflict with each other
- *Architecture* (mandatory) - either specifies a concrete architecture, such as *i386*, or *all* to denote an architecture-independent package
- *Essential* - if set to yes, the package should never be removed, although it can be upgraded or replaced
- *Installed-Size* - specifies the size of the package when it is installed
- *Maintainer* (mandatory) - specifies the person responsible for maintaining the package
- *Description* (mandatory) - a description of the package

The rest of the package metadata specifies the dependencies on other packages as well as conflicting packages using the following fields:

- *Depends* - before package *A* is configured, package *B* has to be configured first. This is a run dependency meaning package *A* cannot run without *B*
- *Recommends* - similar to *Depends*, however package *B* is not necessary, only recommended. If package *B* is not present, *A* may malfunction
- *Suggests* - similar to *Depends*, however *A* can function correctly without *B*
- *Enhances* - opposite of *Suggests*
- *Pre-depends* - this is an installation dependency. Package *A* cannot be installed if package *B* is not already installed
- *Conflicts* - opposite of *Depends*
- *Replaces* - a way to resolve conflicts. If package *A* replaces package *B*, then package *B* will be removed if *A* is installed

DEB Package dependencies can be scoped using the following operators: $=$, $\leq$ and $\geq$.

Packages can also depend on *virtual packages*. Virtual packages do not physically exist, but they can be provided by other packages using the *Provides* field. Virtual packages are not versioned.

Source packages The information presented above is only relevant to binary DEB packages. Source packages are different in several aspects. Firstly, many binary packages can be built from a single source package. Secondly, source packages only have build-time dependencies. Namely, a source package can build-depend or build-conflict with other packages.

Versioning A version number of a DEB package consists of three parts: the *epoch*, the *upstream version* and the *revision*. Epoch is a single integer and the most important part of the version - a package of epoch *n+1* will always be of higher version than a package of epoch *n*.

Sections Each DEB package belongs to a section (*main*, *contrib*, etc.) depending on whether or not they conform to certain coding conventions, as well as what kind of packages they depend on. This allows to assure certain quality of packages in each of the sections.

3.1.2 RPM

Like DEB, RPM (Bailey 2000) packages also exist in either source or binary form. Here we only describe the binary ones.

RPM packages have their own binary file format - same for both source and binary packages. It was designed to be independent of the target machine's architecture, therefore it follows the byte ordering defined for the internet. This format can be divided into the following logical sections:

- *Lead* which contains the package format signature and some information concerning the structure of the package
- *Signature* which is a collection of digital signatures for cryptographic purpose
- *Header* which contains the package metadata, such as package description, dependencies etc.
- *Payload* which is the actual archive with the package content

The header section is where package metadata is stored. The main elements of this metadata are the following:

- *Name* specifies the name of the software being packaged
- *Version*, *Release* and *Epoch* specify the version of the software
- *Description* and *Summary* describe and summarize the content of the package
- *Group* provides a way to organize packages. Groups can be hierarchical.

In RPM, the package headers can also contain information on package dependencies, which allows to establish relationships between packages. Package dependencies are specified using the name of the package and sometimes arithmetic constraints, which is possible because RPM package versions are totally ordered. Available comparison operators are the following: $<$, $\leq$, =, $\geq$, $>$.

Here are the dependency-related tags that an RPM header may contain:

- *Requires* specifies a package needed by the given package to work. The required package can be installed “on the fly” and thus not have to be present before the installation process of the current package starts

- *PreReq* similar to *Requires* but the package has to be present before the installation of the current package starts

- *Conflict* specifies the packages that must not be installed in order to make the current package work

- *Provides*, often referred to as a virtual package, allows a package to specify a capability that it provides. This can also be a way to group packages together

- *Obsoletes* specifies the packages that are made redundant by the current one. Redundant packages are automatically deleted

Versioning A version number of an RPM package consists of three parts: the *epoch*, the *version* and the *release*.

3.1.3 Summary

RPM and DEB are two very common and highly advanced packaging formats for Linux. Therefore, analysing them helps to better understand the requirements that software packages need to fulfil, especially in terms of metadata. It is clear that these two packaging formats share many similarities and the only important differences are the following:

- *Package priority* is a feature that allows to specify how important a package is. It is present in DEB, but not in RPM

- *Essential package* is a feature that allows to specify a package that can never be removed. Again, this feature is available in DEB but not in RPM

- *Multiple versions of a package* - in RPM is it possible to have several versions of a package installed simultaneously. This feature is not available in DEB

Being essentially software packaging technologies, RPM and DEB do not address the distributed deployment issues. Furthermore, neither of the two systems attempts to establish a link between the physical packages and the applications contained within them. This results in package management systems built on top of these formats (such as the Synaptics system) being often unable to correctly manage software uninstallation or handle application conflicts. Finally, since RPM and DEB are only packaging systems, they do not provide the necessary reflexive component model for building autonomic management systems.

3.2 Java JARs & .NET assemblies

JARs and assemblies are packaging formats for the Java and .NET platforms respectively. We describe both of those formats in this section, because they share many similarities—they contain a machine-independent, partially compiled code. This code is compiled into machine language at runtime, just in time (JIT).

Java's machine independent code, the *bytecode*, is executed by the Java Virtual Machine (JVM). Initially, Java bytecode was designed to support only the Java programming language. Currently some other programming languages can be compiled into it. .NET's equivalent of bytecode is called the Common Intermediate Language (CIL). It is run by the .NET's Common Language Runtime (CLR), which can be seen as .NET's equivalent of the JVM. As opposed to Java's bytecode, .NET's CIL was specifically designed to support various programming languages, such as C#, Java, etc.

The packaging format for the Java bytecode is called JAR (Java ARchive). JAR packages contain Java classes compiled into the bytecode and metadata in a form of a manifest file. .NET assembly is a packaging format for the .NET platform. Just as Java JARs, .NET assemblies contain the compiled code, a manifest file and possibly other resources. A simplified diagram of both of those packaging formats is represented by figure 3.1.

The manifest files, a purpose of which is to describe the content of the packages, have different formats in JARs and assemblies—they are regular text files in Java JARs and XML files in .NET assemblies.

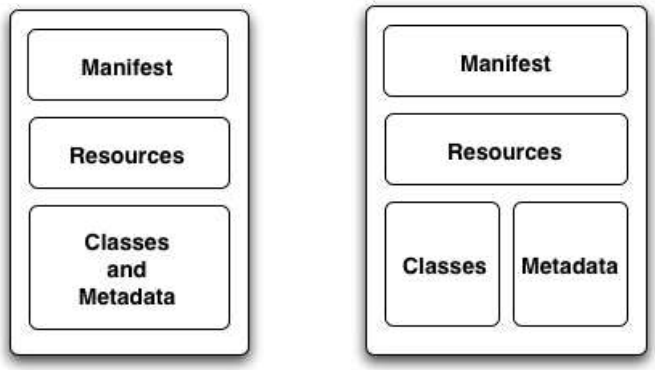


Figure 3.1: Java JARs and .NET assemblies

JARs do not provide a versioning scheme, neither do they allow for explicit declaration of dependencies on other JARs. Compared to assemblies, they are very simplistic in this respect. Assemblies can be defined as versioned, self-describing binaries (DLL or EXE) which contain a collection of types such as classes, interfaces, structures etc. An assembly can span multiple files (the multi-file assemblies), but also a single file can contain one or more assemblies. In a typical case an assembly consists of a single DLL or EXE file. Despite the fact that assemblies can have the DLL extension, they are different from the old (C-style) DLLs, the only similarity being a conceptual one—both are libraries of code intended to be loaded and called by other programs. The EXE assemblies are executable programs, but they can also export DLL resources to be used by other programs.

.NET assemblies can be strong- and weak- named. Weak named assemblies do not have any restrictions on their naming convention. Strong named assemblies have to specify: a name, a version number, a programming language used to develop them and a public key. This approach is extremely useful for version identification. Compared to Java, in which the only versioning scheme is the fully qualified class contained within a JAR, strong named .NET assemblies have a much more powerful versioning scheme. This, coupled with the notion of application domains, allows .NET applications to simultaneously use several versions of the same classes. This feature is not easily achieved in Java—it can be obtained by clever usage of *class loaders*, the entities that convert bytecode into the code executing within a JVM. This will be explained later in this document.

The notion of an application domain in .NET does not exist in Java. Each application domain is an isolated execution environment and communication between different application domains must be an inter-process communication – just like Remote Method Invocation (RMI) in Java. Assemblies loaded in one domain are not available in the other domains. Each application domain manages its own assemblies.

Class loading is an important difference between the Java and .NET. The first major difference is that .NET is only able to load CIL code from assemblies. The Java class loaders, by default, can load classes either from JAR files, or directly from .class (bytecode) files. Moreover, one can define his own class loader capable of loading code from yet another file format, over the network etc.—this feature is not available in .NET. On the other hand, .NET allows for dynamic unloading of assemblies, which in turn allows for the evolution of the applications deployed on .NET without the need to restart the platform, a feature not available in Java. However, only complete assemblies can be unloaded.

3.3 Summary

In this chapter we have described several popular software packaging technologies. We were interested in these systems because packaging can be considered the lowest layer of deployment thus needs to be addressed by any system attempting to provide a full-featured deployment functionality.

RPM and DEB are popular packaging formats in the Linux/Unix environments. They share many similarities and provide an extensive set of metadata that the package provider can use in order to express package dependencies, conflicts, priorities etc. The two packaging formats support source and binary packages—source packages are independent of the target machine's architecture, binary packages contain code that was compiled specifically for a given architecture. The main drawback of these systems is that they can only be used to install software on Linux/Unix machines and are not compatible with other operating systems. Furthermore, they are not coupled with a component model, thus make building autonomic management tools on top of them impossible.

Java JARs and .NET assemblies are packaging formats for the Java and .NET platforms respectively. These platforms are modern, machine-independent technologies that use a form of intermediate code (bytecode in Java, CIL in .NET) within the software packages. This allows the packaging to be independent of architecture of the target machines as well as the operating systems running on them. JARs are only units of grouping for the precompiled Java code—they do not allow for expressing dependencies or conflicts between this code. Furthermore, they do not provide metadata to describe the content of packages. .NET assemblies are more similar to RPM and DEB packages in terms of

completeness of the metadata and thus are more advanced than Java JARs. Yet, .NET is limited in terms of dynamic loading of code, as opposed to the Java platform, which makes building custom component-based systems on top of it difficult.

Résumé de chaptire 4

Dans le chapitre 4 de cette thèse on présent plusieurs solutions visant a résoudre le problème de la manque de support pour la modularité dans le langage Java. On a choisi Java comme environnement car c'est un langage type, oriente objet, dans lequel des nombreuses modelés a composants était implémenté. Par ailleurs, plusieurs plateformes de déploiement décrit dans le chapitre 2 de cette thèse sont construits en Java et par conséquence souffrent a cause de limitations de cette langage et son environnement d'exécution—la JVM (Java Virtual Machine). Ensuite, comme Java est une technologie relativement moderne et en cours d'évolution, elle est construit a partir des années d'expérience dans plusieurs autres langages de programmation, elle partage également des nombreuses similarités avec d'autres langages modernes oriente objet, tel que le C#, et leurs environnement d'autres environnement d'exécution, tel que .NET. Par conséquence, les problèmes de modularité décrit dans ce chapitre et les approches qui essayent de les résoudre sont également applicable aux autres technologies a partir lesquelles les plateformes a composants et leur systèmes de déploiement sont construit.

Chapter 4

Module systems for Java

Contents

This chapter presents several solutions to the lack of modularity in the Java programming language. We have chosen Java, because it is an object-oriented, typed programming language in which many component models are implemented. Furthermore, many of the existing deployment frameworks described in Chapter 2 are built in Java, thus they suffer from the limitations of this language and its execution environment—the Java Virtual Machine (JVM). Finally, since Java is a relatively modern and evolving technology, it not only builds on many years of experience in various programming languages, but also shares numerous similarities with other modern object-oriented languages, such as C#, and their execution environments, such as .NET. Thus, modularity issues discussed in this chapter and the solutions that attempt to address them are in many aspects applicable to other existing technologies on which component-based frameworks and their deployment solutions are built.

Before discussing the modularity in Java, we need to briefly present its encapsulation and scoping, as well as type system and linking mechanism. A more in-depth description of these aspects of Java will be given in the contribution part of this document.

Java programming language provides classes and packages for static encapsulation and scoping. Packages, such as *java.lang* define name spaces for Java classes. Different visibility modifiers can be applied to classes, their fields, methods etc. such as *public* or *private*.

Class loaders, on the other hand, can be seen as dynamic name spaces. A class loader is an object called by the JVM whenever a reference to a name (the fully qualified class name—FQN, thus the name of a class plus the name of its package) needs to be resolved. The class loader converts the Java bytecode, that it finds either in the *CLASSPATH* in a file system or downloads over the network, into a *Class* object. From the class objects, object instances can then be created. The *Type* of each object instance is defined as a tuple, containing the fully qualified class name of the class from which this object was created, and the class loader that converted this class' bytecode into a class object. Thus, class loaders in Java can be seen as name space mechanisms. Since Java allows the class loaders to delegate the loading of a classes between them, one can build sophisticated module mechanisms on top of class loaders.

As will be presented in this chapter, the static scoping combined with class loaders is insufficient to provide the true modularity in Java. The research projects described below attempt to build higher-level, reusable module frameworks for the Java platform.

4.1 OSGi

The OSGi Alliance has created a set of specifications defining an environment for the deployment and execution of component-based, service-oriented applications on top of the Java platform (*Open Services Gateway Initiative, OSGi service gateway specification, Release 4* 2005). Initially these specifications were targeted at the set-top box devices, on which different vendors could remotely deploy and manage software services written in the Java language. Since then, OSGi has evolved significantly and is now used in various contexts, spanning from mobile phones to server-side Java EE platforms. OSGi can therefore be seen as a general, component-based environment for the Java applications executing within a single JVM. It is thus very interesting to see how it copes with the problems of modularity, versioning and dynamic updates in Java and analyse its shortcomings.

Essentially, the OSGi specification can be divided into the (1) physical part and the (2) runtime part. The physical part is about packaging and deployment of OSGi components, whereas the runtime part is about managing life cycle of these components and the dependencies between the services over which they communicate.

4.1.1 Physical part

The physical part of the OSGi specification defines a packaging format for the OSGi components, called *bundles*. This format is based on Java jars enhanced with metadata stored in a manifest file. There is one manifest file per bundle. Bundles' metadata is processed by the OSGi platform when bundles are loaded into the OSGi execution environment. The main element of bundle's metadata is the declarations of dependencies. Each bundle can specify what Java packages it provides (exports) and what packages it needs (imports). Java package names, not class names, have therefore been chosen by the authors of OSGi as the granularity of import/export dependencies. In the metadata, bundles can express additional information on the provided/required packages, such as versions. Furthermore, the metadata file lists the directories and JAR files which constitute the given bundle's class path and which are contained within the bundle archive (therefore an OSGi bundle JAR file can contain nested JARs). A sample manifest file of an OSGi bundle is presented in figure 4.1.

```
Bundle-Name: SampleBundle
Bundle-SymbolicName: fr.jade.bundles.sample.SampleBundle
Bundle-Description: A sample bundle
Bundle-Version: 1.0.0
Bundle-Vendor: SARDES
Bundle-Activator: fr.jade.bundles.sample.Activator
Bundle-Classpath: .,/classes,/lib/jms.jar
Import-Package: fr.jade.bundles.library;specification-version="1.0.0"
Export-Package: fr.jade.bundles.sample.interface;specification-version="1.0.0"
```

Figure 4.1: A sample OSGI manifest file

When an end-user attempts to deploy a bundle into the OSGI environment, the environment attempts to resolve the bundle's dependencies. If some of the bundle's dependencies can not be resolved, its exported resources will not be available to other bundles. If the resolution phase succeeds, the platform will be able to proceed and start the bundle.

Starting a bundle means invoking its activator class. This class is specified in the bundle's manifest and is the entry-point to the component. Bundles are not required to declare an activator. If they don't, the bundle can not interact with the runtime part of the platform, thus its only purpose is to serve as a library of compiled Java code. If a bundle declares an activator in its manifest, the activator class must implement the *org.osgi.framework.BundleActivator* interface. This interface is used by the OSGI framework to manage the bundle and give it the access to the OSGI platform's runtime.

4.1.2 Runtime part

When a bundle is resolved, thus its dependencies are satisfied, the OSGI runtime calls a *start* method of its activator, passing an instance of a *BundleContext* as this method's only argument. This call gives the programmer of the bundle the reference to a *BundleContext*—an interface through which bundles register services that they provide and lookup services that they require. Each bundle can provide and use as many services as it wants.

OSGI services are published in the bundle context by specifying the service interface, the reference to an object implementing this interface and finally a set of meta-information about the service. To lookup a service in the bundle context, one has to define the service interface and an LDAP-like query against which the provided services' metadata will be matched.

The OSGI specification imposes the organisation of class loaders within the framework—there is always one class loader per bundle. A good example is the Felix platform. Its underlying class loading mechanism, built on top of the *Module Loader* (Hall 2004) extensible class loading framework, follows precisely this requirement—each bundle installed within Felix is equipped with its own class loader. When resolving the bundle accordingly to its import/export dependencies, Felix will configure the class loader delegation graph. Furthermore, the platform's underlying class loading mechanism observes the lifecycle events of bundles as they come and go and reconfigures the class loader graph accordingly. As a result, often an update of a bundle imposes that bundles which transitively depend on it be also updated. An update means simply recreating the bundle's class loader, thus reloading all the code from it. It is necessary because Java class loaders cannot unload classes, yet they define types.

4.1.3 OBR

Finally, an interesting aspect of the OSGI specification is the OSGI Bundle Repository (OBR). OBR allows the users of OSGI to store their bundles, including metadata, on an HTTP server, from which they can be seamlessly deployed on target machines. When a manager of an OSGI gateway asks the OBR to deploy a bundle, automatically this bundle's dependencies will be analysed and the bundles missing on the given gateway will also be installed.

4.1.4 Summary

OSGI is currently the most advanced module system for Java applications. It handles not only the versioning of Java software components within a single JVM, but also allows for dynamic installation, removal and updates of those components. It comes with a rather lightweight API, thus writing Java software compatible with OSGI is relatively straightforward and the existing Java applications can fairly easily be ported to OSGI-compatible versions and benefit from the deployment and service-oriented aspects of this framework.

Furthermore, OSGI comes with a growing library of existing software components which can be used off-the-shelf. It also has a powerful tooling, including the Eclipse plugins to easily write, compile, store and deploy bundles. This is important because those tools ensure that, for example, Java types used within OSGI bundles are resolved against the same dependencies during development (compilation) and execution—thus eliminating the problems commonly encountered with Java class path. Those advantages have made OSGI especially popular with the developers of Java-based containers of third-party applications (such as Java EE servers), which often suffer from unexpected type resolution problems.

Despite the above advantages, OSGI also presents several important limitations. First of all, it cannot be considered a real component framework. There is no notion of explicit, reflexive software architecture in the OSGI specification. Service-level dependencies between bundles are hidden within the activators, and it is the bundle developer's responsibility to correctly manage those dependencies. This is not only against the definition of a component as a piece of software which is bound to other pieces of software by a third-party entity, but also can quickly lead to bugs such as non-nullified references which prevent the unused code from being garbage-collected. Therefore, building architecture-based deployment or autonomic management tools on top of OSGI is difficult.

4.2 Java EE servers

Java Enterprise Edition (EE) (*J2EE: Java 2 Platform, Enterprise Edition* 2002) is a platform for building server-side applications in the Java programming language. It extends the "standard" Java platform in that it provides a set of libraries that facilitate the construction and deployment of transactional, distributed, fault-tolerant and scalable multi-tiered applications. Each application written in the Java EE technology is component-based and deployed within the Java EE container, according to the JSR88 specification (see Section 2.3). This specification, however, does not address the issue of modularity and versioning of Java EE application components within the Java EE containers.

The problem of modularity is inherent to the Java EE containers for two main reasons:

- The need for isolation of the application components (EJBs, Servlets/JSPs) from one another
- The need for isolation of the Java EE containers internals from the application components

4.2.1 Isolation of the Java EE components

Java EE components such as the EJB business components or Servlet/JSP presentation components can be developed and deployed in the Java EE container by various providers. This means that the Java EE container (the server) has to assure that components deployed by multiple actors are properly isolated and that no conflicts occur when those components are deployed and when they execute.

4.2.2 Isolation of the container from the applications

The Java EE container itself is usually written in the Java language. The role of the container is to provide all the non-functional services, such as transaction, security, fault-tolerance, messaging, logging etc. to the application components. Most of the existing containers reuse legacy libraries to provide those non-functional services. For example, the JOnAS Java EE server uses the JORAM (*JORAM: Java Open Reliable Asynchronous Messaging* 2002) JMS (*Java Message Service Specification Final Release 1.1* 2002) library to provide communication services to Java EE applications and the Monolog library to provide the logging. The code of these services and the JOnAS server should be isolated from the code of the application components deployed within the server. If it is not, the applications may fail to work correctly after it is deployed in the server due to the clashes between the Java classes contained within the applications and the ones used by the server.

The two types of isolation presented above are difficult to achieve in Java, because as explained in the introduction to this chapter, Java has no support for modules—it uses JARs as packaging format, which do not support versioning, and class loaders for modularity, which are too low-level. The JSR88 specification does not attempt to address this issue. As a result, providers of the Java EE servers use ad-hoc solutions to the problem of modularity in the Java environment. All of those solutions are built on top of the Java class loaders, but vary in terms of approach (*Understanding J2EE Application Server Class Loading Architectures* 2002).

4.2.3 JOnAS

Most of the existing Java EE servers use a tree-based hierarchy of class loaders to obtain the necessary modularity (*WebLogic Server, Application Class loading* 2004) (*WebSphere Software Information Center, Class loaders* 2003) (Bailliez 2005). This is illustrated by the JOnAS server (*JOnAS: Java Open Application Server* 2005) and its class loading architecture depicted in figure 4.2. In this hierarchy, the children class loaders always delegate the request of loading a class to their parent before they attempt to load the class themselves. Class loaders which are not on the same branch are separate namespaces for classes. In the case of JOnAS, the *System* class loader loads all the “standard” Java libraries as well as the Java EE API. The *Commons*, *Tools* and *Application* class loaders load the classes that build the JOnAS Java EE server.

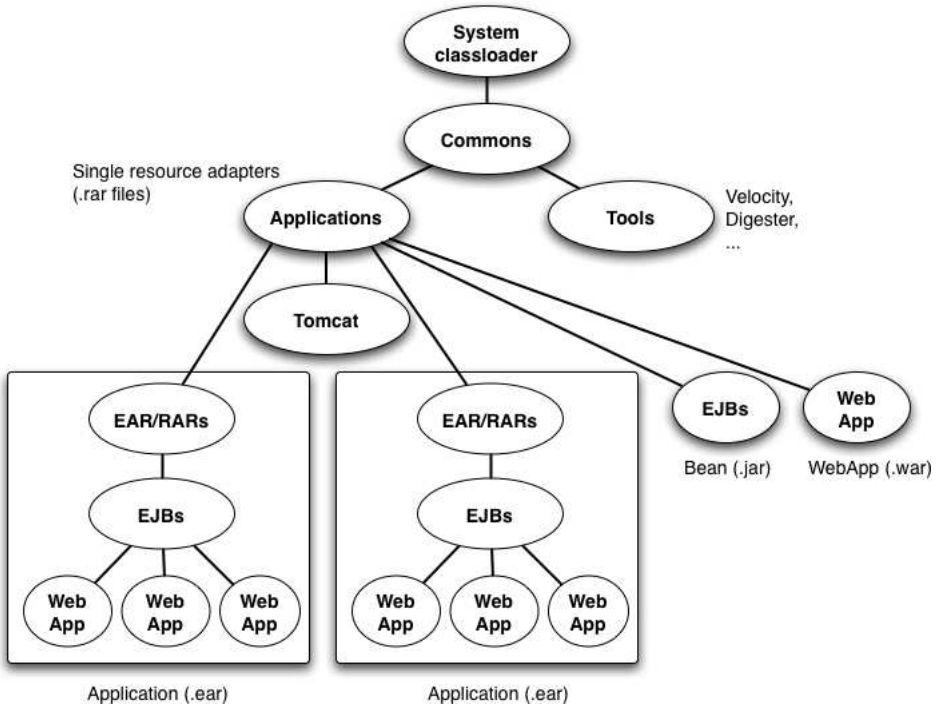


Figure 4.2: Hierarchy of class loaders in the JOnAS Java EE server

For each Java EE component deployed within the JOnAS Java EE server, the server creates a proper class loader, which is always a child of the *Application* class loader—that way the application components can "see" the Java and Java EE APIs. How those children class loaders are created depends on how the application components were packaged, as is visible in picture 4.2 for the Enterprise Application Archives (EAR) and for "stand alone" EJB jars and WAR files. The server also manages how the application archives (EJB jars, EAR and WAR files) are stored in the file system once they are deployed in the server.

Having such a hierarchy of class loaders partially solves the issue of isolation of Java EE components—depending on how the components are packaged, each EAR is a namespace for its EJB jars and WAR archives, or each EJB jar and WAR archive can be a namespace of its own. The solution is partial for several reasons. Firstly, if the application components are not contained within the same archive, they can only communicate using serialization. This has an important performance overhead and eschews the Java type system. It would be better to be able to define fine-grained import/export dependencies between those application components. Secondly, how the applications were packaged has an impact on the reconfiguration capability—only complete archives can be redeployed, thus it is impossible to update a single EJB from Enterprise Application without redeploying the whole application.

In terms of isolation between the container and the applications deployed within it, the above solution does not work. Since the class loaders created by the container for the application components deployed within it inherit from the class loader that load the server's code, the application components' class loaders "see" all the classes that the container consists of. This has two important implications: (1) the classes contained within the application can clash with the ones used by the server. This

means that the applications deployed within the container can have unexpected behaviour. And (2) the applications can access the internals of the server and for example call the `public static` methods of the server's code.

4.2.4 JBoss

Another approach to the problem of modularity within a Java EE container is the one adopted by the JBoss server (Fleury and Reverbel 2003). It introduces the concept of a *class loader repository*, with the *unified class loader (UCL)* being the centeral piece of the class loading mechanism. Each UCL is associated with a class loader repository of classes and resources—UCL can only have one repository, but many UCLs can use a single repository. Every time a class is loaded by a UCL, it becomes available via its associated repository. This way, all the UCLs associated to a given repository share classes.

By default JBoss has a flat class namespace because it uses a single repository. This means that every UCL "sees" all the classes loaded by other UCLs. Two things are important: (1) one can configure the delegation scheme, namely either a UCL first looks for classes in the repository, and only if it doesn't find them there tries to load them itself, or it first looks for classes "locally", which means that it breaks the default delegation scheme defined by Java. (2) when a UCL looks for classes in the repository, the order in which other UCLs were added to this repository becomes important, because if the class is not found in the repository the UCLs are queried for a given class in the order in which they were added to the repository.

In order to handle more complex isolation scenarios than the one where the class namespace is flat, JBoss supports hierarchies of class repositories, with user-defined delegation schemes between them.

In general, the approach to code isolation taken by the JBoss server is more complex and more powerful than the one adopted by most of the other Java EE servers. It provides a bigger flexibility in defining hierarchies of class loaders, but at the same time requires more knowledge on the subject. Furthermore, the authors of JBoss do not attempt to generalize their solution beyond the context of the Java EE application components. Neither do they attempt to integrate it into their implementation of the JSR88. As a result, the deployment of application components within the JBoss server is orthogonal with respect to the configuration of the isolation between these components at runtime. A more unified approach to the subject would be interesting.

4.2.5 Summary

The need for modularity in Java is especially visible in the context of component containers, such as the Java EE servers. Due to the lack of an existing standardized approach, providers of the Java EE servers implement their own ad-hoc solutions. Most of the approaches are based on the "standard" tree-like hierarchy of Java class loaders. Such approach makes reuse of deployed code difficult, thus implies a high level of redundancy, which has impact on the memory consumption. Moreover, it makes the server classes visible to application components, which can result in name clashes and opens a security hole for malicious behaviour of components.

The JBoss server follows a more complex and flexible approach which consists in using a federation of class loader repositories and user-defined delegation schemes between them. This solution is still insufficient in that it is not general and is not integrated with the JSR88.

The context of Java EE servers is a good illustrations of the need for modularity in the Java environment as well as of how ad-hoc solutions are insufficient in this area. At the time of writing of this document, several Java EE servers, including JOnAS (Desertot et al. 2006), IBM WebSphere and BEA WebLogic have been or are being ported to the OSGI platform in order to address the class loading problems in a standardized manner.

4.3 iJAM & JSR277

iJAM (Strnisa et al. 2007) is essentially a formalized module systems built on top of the JSR277 specification. The JSR277 (*Java Module System (JSR277)* 2004) is a response to the lack of modularity in Java from the Java Community Process (JCP).

JSR277 provides an architecture for the development and deployment of module-based Java applications and libraries. In that respect, it is similar to OSGI. Authors of the JSR277 argue that the JAR packaging format, which was supposed to be a distribution and execution format for Java, did not scale particularly well in either of those roles, mainly due to the lack of support for versioning. This leads to the already mentioned problems with the classpath and in general, to the so called JAR Hell.

To address these problems, the JSR277 provides the following:

- A distribution format for the Java modules. This distribution format covers the packaging and metadata of units of delivery for collections of Java classes. Metadata contains information about the module, classes and resources within the module and the dependencies on other modules. Module's metadata also defines the list of classes exported by the module.
- A versioning scheme for modules which covers the version of a module as well as the versions of its dependencies
- A repository for storing, discovering and retrieving modules with support for module isolation
- Class loader based runtime support for discovering, loading and integrity checking of modules

The specification makes a distinction between *module definitions* and *module instances*. Module definitions are the units of reuse, packaging, versioning and deployment in the module system. The definitions specify module's imports and exports and they are stateless. Module definitions have a physical representation in a form of a *module archive*. Module archives contain metadata, classes and resources that constitute the module. From the physical packaging point of view, module archives are called JAM files and are essentially JAR files with a module-specific manifest. Each module archive is self contained and does not reference external resources. There is always a one to one correspondence between a module archive and a module definition.

From the module definitions one can instantiate module instances at run time. Multiple instances of a module definition can coexist in the JVM, however to maximize sharing a single instance should be reused. Each module instance has a copy of classes and other resources contained within the

module, it can also be interconnected with modules which satisfy its imports. Thus, each module instance is a namespace at runtime.

The JSR277 provides a detailed specification of how modules are supported by the JVM at runtime. There is a one to one association between a module instance and a module class loader.

Module definitions and instances are reified by an explicit module API. This allows for the programmatic creation and introspection of module definitions in the repository and for introspection of module instances at runtime.

Since the JSR277 is mainly a textual specification, the iJAM projects attempt to formalize it. Based on the formalization of JSR277, the authors of iJAM attempt to provide a reference implementation of the specification.

In order to formalize this JSR the iJAM project defines a formal language called the LJAM, verifiable in the Isabelle/HOL automatic proof assistant. The authors' conclusion after formalizing the JSR277 is that the specification has limitations that make solving certain basic software engineering problems impossible. Furthermore, they argue that it does not fully solve the issue of class name clashes. They propose an alternative that solves these limitations.

Summary The JSR277 is an attempt to provide a standardized module system for the Java applications supported by the Java Community Process. It can be seen as an alternative solution to what OSGi provides. This JSR goes further than the OSGi specification in specifying the class loading mechanism underlying the module system, it also provides detailed information on what module repositories are, which is not the case in OSGi. However, the main drawback of this JSR is that it does not address the dynamicity. Modules are static units of code isolation within a JVM. If one wants to dynamically remove or update modules, the JVM needs to be restarted. This is not an acceptable solution in an environment that requires hot (re)deployment of components instantiated from modules, such as Java EE containers. In this respect, OSGi is a much more advanced solution. Furthermore, the JSR277 is not coupled with a component or service oriented application execution model, thus it is difficult to build architecture-based deployment and software management frameworks on top of it.

The iJAM is a research project providing a formalization of JSR277 and a reference implementation of the module system based on this formalization. It identifies other shortcomings of the JSR277, especially in terms of solving class name clashes. iJAM attempts to address these shortcomings by defining its own class loading delegation scheme. However, the solution proposed by iJAM can lead to yet another types of problems, as explained in (*Formalized Class Loading* 2007).

4.4 MJ

MJ (Corwin et al. 2003), or a "Rational Module System for Java", is an IBM project which tries to address the lack of modularity in Java. Work on MJ within IBM was partly triggered by the experience with large Java-based containers for Java software components, such as the WebSphere Java EE server.

According to the authors of MJ, when constructing large software systems, it is desirable to decompose the system into collaborating components with well-defined interfaces between them. Each component should specify declaratively, using mechanisms that can be statically checked and dynamically enforced, what functionality of other components it depends on and what functionality it

makes available to other components. Moreover, according to the authors of MJ components must be hierarchical—a component must be able to contain an arbitrary number of other components. Internal components may either be hidden or exposed as part of the containing component's interface.

Authors identify that building such system in Java is difficult, especially because there is no distinction between *linking*, which is the process of resolving the textual reference to a name found in a piece of code which is part of the current component, and the *component activation* which is the process of resolving a reference to a name imported from another component. This causes serious problems in applications that need to support multiple versions of the same class. Furthermore, authors argue that Java packages are only a static name space mechanism and class loaders are too low-level. As a result, Java applications often suffer from unexpected interactions between independent components deployed within a single JVM. Furthermore, the class path against which Java program is compiled can very well be different from the class path against which the same program is executed. This can result in *NoClassDefFound* errors when the class used at compilation cannot be found at execution, or in the usage of a wrong version of a class at execution time—possibly a completely different class than the one used for compilation, but which just happens to have the same fully qualified name.

Since Java allows for defining of custom class loaders, the above mentioned issues result in the proliferation of ad-hoc, class loader based solution. These solutions, combined with elaborate uses of the class paths, attempt to address the lack of a sufficiently strong module system. However, the implicit dependency between the compile-time and the runtime class path is a particular problem, not addressed by most of the custom solutions.

To address the above issues, authors of MJ have developed a complete module system for Java. The system comes with (1) a module description language, (2) a module repository which can be used for both, compilation and execution, (3) a high-level API for the manipulation of modules and (4) a class loader based execution environment for MJ modules.

The MJ module description language is the essential part of the MJ system from the end-user point of view. It completely replaces the need for specifying a class path to the JVM and the need for creating custom class loaders. Instead, MJ comes with a *component registry*, which contains both the module metadata and module provided classes (bytecode). The module's metadata contains information on:

- What classes the module provides and where they are stored
- What other modules this one depends on
- Which classes are made visible to other modules, which can be sub classed and which package-prefixes are restricted
- Initialization code which is called when the module is started

There is one module description file per each module. An example module description is illustrated by figure 4.3.

Modules can define *load* and *main* methods. The load method is called whenever a module is first loaded by the MJ system. The main method is called when a module is started from command line.

```
provides "catalina.jar";

import * from xerces;
import * from bootstrap;
import com.sun.tools.* from tools;

hide * in *;
export org.apache.catalina.* to webapp;
export org.apache.catalina.servlets.* to servlets;

forbid org.apache.catalina.* in *;

module catalina {
  public static void load() {
    System.setProperty("javax.xml.parsers.SAXParserFactory",
      "org.apache.xerces.jaxp.SAXParserFactoryImpl");
  }

  public static void main(String[] args) {
    org.apache.catalina.startup.Bootstrap.main(args);
  }
}
```

Figure 4.3: A sample description of the MJ module

Before a module can be started, it first needs to be stored in the *repository*. This is performed using a tool provided with MJ—the *modjavac*. This tool takes a module description file, parses it and creates a corresponding module in the repository. Next, Java source code can be compiled against a module, using the *javamodc* tool. What the javamodc actually does, is the checking of static constraints and a proper configuration of the compilation class path. Finally, one can start the application from a command line by specifying a repository and a module to start.

The MJ system defines a set of specific class loaders used by the framework when executing an application. Each MJ module is associated with exactly one class loader instance. The module and its class loader are named using a unique ID. Module's class loader loads and defines classes provided by the module. Any inter-module class loading is delegated to a central unit, called the *component class loader*. This class loader uses the information from the component registry to verify if the delegating module has the right to load classes from the module to which it delegates. Clearly, this approach prohibits the users of MJ from defining the custom class loader. Instead, users can add module dynamically into the MJ system using the MJ's API.

Summary MJ is an interesting example of a module system for Java. It is complete in a sense that it provides a configuration language, a module repository, an execution environment and finally the tools to efficiently create, compile and execute modular Java code. In this respect, MJ solves the issues of unexpected code interactions and incoherence between build and execution class paths found in "standard" Java. Moreover, it provides a powerful mechanism for exposing different subsets of module's content to various importers.

However, MJ needs to be seen mainly as an isolation platform for the Java code, not a truly dynamic platform for Java components. MJ is not component-based in a sense that MJ modules are

not reflexive and cannot be introspected. Thus, autonomic managers would be hard to build on top of MJ. This is even more true due to the fact that MJ does not provide an API for the undeployment of modules. In fact, the module life cycle in MJ is very limited and the only way to remove modules is to restart a JVM.

Finally, the notion of import/export dependencies in MJ is simplistic in that it is based on the names of modules. A better solution would be to have a more flexible language for describing the dependencies, including their versions, similar to what is provided in OSGI.

4.5 JPloy

JPloy (Luer and van der Hoek 2004) is a "user-centric" deployment platform. By that, the authors of JPloy mean that they give more control over the deployment process to the users of the software, rather than to the software manufacturers.

Their motivation is that with the increased usage of fine-grained components, the deployment activities, such as installation and configuration, becomes a continuous operation. This is mainly due to the fact that updates for individual components are frequently available.

Authors of the JPloy project argue that currently there are two approaches to the deployment and composition of component based software: the component platforms and deployment tools, with no common ground between the two. According to the authors component models allow the user to configure an application at their site, but the process of installation and configuration is manual and error-prone. Deployment tools, on the other hand, support the installation of components, but they do not support their configuration – rather they assume that components are configured at the manufacturer's site. The authors argue that there's a need to blend the two approaches in order to obtain a user-centric deployment technology for a component platform. Authors identify the following requirements for a user-centric deployment platform:

- *Interference-free deployment* means that installation and configuration of components should not change the behaviour of these already installed. A special case of interference-free deployment is concurrent deployment of several versions of the same component.

- *Independent deployability and absence of strict dependencies* this requirement states that there should be no strict dependencies between components since they are independent units of deployment. In practice, a component's requirements should be possible to resolve by several components, not just a specific one. Thus the component platform should provide mechanisms to specify dependencies not based on component identities. Independent deployability means that components are deployed independently of the application—once components are deployed, the application builder at user's site decides how they are configured and assembled. Therefore, the information on component interactions (assembly) and configuration is stored elsewhere than in the components themselves.

- *Compatibility with legacy code* means that the component platform has to provide an effortless way to deploy legacy code—by, for example, wrapping the legacy software to make it compatible with the platform. It may be possible to create such wrappers automatically.

Authors also state *transparent updating* and *incremental builds* as interesting features of the deployment platform. The former is a mechanism of automatic update of components when their new version is released, which would be interesting for applications build using a large number of fine-grained components, the latter means that the deployment platform allows for deployment and execution of only parts of the application, for the purpose of testing etc.

Authors of the JPloy framework identify several ways to provide support for component deployment:

- By integrating it into the programming language. This approach has the disadvantage in that the deployment information is lost after compilation. Since components are usually shipped in a compiled form, programming-language level support would not be helpful for the application builder.
- Using external tools to modify existing components by editing their binary representation. This approach has the disadvantage of creating many binary versions of a component.
- Extending the platform's API and dynamically injecting the calls to the platform into the component binaries. It may be hard to apply on legacy components which were not specifically designed for a given platform.
- Extending the platform's loading and linking mechanism. This is the approach followed by JPloy.

Authors considered either .NET or Java platform for implementing their approach. They have chosen the latter mainly due to the exhaustive documentation on its extensible loading and linking mechanism—the class loader. Their solution extends the Java class loader so that it can read configuration files and modify the bytecode of component classes when they are loaded into JVM. This bytecode manipulation is performed according to the information obtained from the configuration files. The configuration files have a simple syntax illustrated by an example below 4.4:

```
1.  wren        = c:\wren\wrenclient.jar
2.  argo        = c:\wren\argouml.jar
3.  xerces      = c:\wren\xerces.jar
4.  argoinit    = c:\wren\argoinit.jar
5.
6.  wren main
7.
8.  wren use argo
9.  argo use xerces
10. argo use argoinit
11.
12. argo replace
        org.argouml.application.PreloadClasses
        edu.uci.wren.PreloadClasses
```

Figure 4.4: A sample JPloy configuration file.

This language allows to:

- Define components—see lines 1-4 in the example. A JPloy component is simply an alias to a jar file

- Define an entrance point to the application, that is the component which provides the *main(String[] args)* method. See line 6 in the example

- Define usage (import/export) relationships between components—see lines 8-10. Usage relationships mean that a component which declares a usage of another component has the access to all the resources contained within the given component

- Replace at load time a class by a different one

The possibility of JPloy to define components and the usage relationships between them solves the problem of name conflicts, and thus addresses the *interference free deployment* requirement. Since every component in this relationship can be replaced by a different, compatible one, the *independent deployability* requirement is also addressed. Finally, JPloy does not require the modification of source code or the repackaging of components, thus it also addressed the *compatibility to legacy components* requirement.

Summary The JPloy framework can be resumed to an extended class loader which allows, through configuration files, to apply custom configurations to existing Java jar files without the need to repackage them. It can therefore be seen as an evolution of the Java class path mechanism. This approach allows for an elegant way of resolving name space and versioning conflicts between components. Moreover, through bytecode manipulation at the load time of components, JPloy allows for replacing some classes within a component, without the necessity to modify and recompile the source code of these components.

However, JPloy has several disadvantages. Firstly, it is Java centric and does not attempt to generalize its approach through higher level modelling of component packaging, loading and dependencies. Moreover, in the Java world a more promising approach to the modularity problem seems to be the one proposed by OSGI or MJ—where many class loaders are used. This approach seems better because by applying the bytecode manipulation, JPloy prohibits the usage of reflection. This is not the case in OSGI or MJ. In JPloy there's also no way to make parts of the components private—a component is either exported as a whole, or not at all. This again is not the case in OSGI or MJ, where exports are on the Java package level. JPloy does not explicitly address the component undeployment issue, which is an important feature in the evolving, component based software.

Furthermore, JPloy does not address the distributed deployment—it has no notion of repositories of software components, thus can only handle components which are already present in the file system of the target machine. Finally, it does not provide a runtime component model, therefore the notion of a component only exists in the JPloy configuration files, but is lost at the execution time of the application, which prohibits any form of runtime management of executing JPloy applications.

4.6 Summary

In this chapter we have described various solutions to the lack of modularity in the Java platform. Those solutions vary in terms of completeness—OSGI is a mature technology used in the industrial context, whereas projects such as MJ, JPloy and JAM are research prototypes.

A common drawback of all those approaches is that they only address the problem of isolation of code, without placing themselves in the context of component-based software. Therefore, modules that they propose are not reified at runtime and are not part of the reflexive software architecture. This makes it difficult to build autonomic systems on top of those solutions. Furthermore, the presented systems do not address the issue of distributed software deployment, thus have a largely limited scope.

Résumé de chaptire 5

Dans le chapitre 5 de cette thèse on présente une analyse comparative des éléments de l'état de l'art décrit dans les chapitres 2, 3 et 4. A partir de cette analyse on peut conclure que la modélisation et gestion des implémentations physiques des composants logiciels est un aspect clé des systèmes de déploiement pour les applications a base des composants. Cette modélisation est nécessaire car les implémentations des composants ainsi que le faon dont elles sont résolu a un impact sur la validité de l'architecture logicielle a l'exécution. Comme on va le présenter dans la deuxième partie de cette thèse, cette modélisation facilite la construction des plateformes de gestion autonome pour les applications a base des composants.

Chapter 5

Summary

In this first part of the thesis we have analysed the existing solutions around software deployment, packaging and modularity. The goal of this analysis was to investigate whether any of the existing solutions provides a complete deployment system which could be used in the architecture-based management context.

Table 5.1 presents our comparative analysis, according to various criteria. By interference-free deployment we understand the possibility to install and configure software components without them having an irreversible impact on other components within the system. A special case of interference-free deployment is having multiple versions of a component running simultaneously. By independent deployment we understand the lack of strict dependencies. This means that dependencies should be expressed on flexible requirements and not on specific components that fulfil them. Transparent updating is a mechanism of automatic deployment of software components if, for example, their new versions are released. Finally, the incremental builds are about configuring partial software architectures, for example for the purpose of testing. Other criteria listed in this table, such as support for dynamic updates, undeployment of components etc. are straight forward. The table illustrates that none of the systems described in the state of the art part of this thesis supports the full gamut of requirements that we have identified. This is not surprising, since our criteria of evaluation concern many aspects of software deployment and runtime management, which is a vast field. However, it is interesting to see what particular categories of systems that we have analysed are missing.

The existing deployment frameworks, such as SmartFrog, DAnCE and others are mainly limited in terms of software packaging and installation. All of those systems support only a single type of software packages. Most of them do not support package installation. Instead, they assume that the implementations of deployed applications are always readily available on target machines. Except for the NIX framework, none of the deployment frameworks provides a repository into which packages can be released, or addresses the issue of dependencies and conflicts between those packages. Except for the JSR88 specification, none of the system addresses the dynamic updates of software. Finally, most of these systems do not provide a notion of runtime components, thus make building autonomic frameworks on top of them challenging. The only exception to this rule is the SmartFrog platform, which we consider as highly similar to the initial version of the JADE platform described in this thesis.

The existing packaging solutions are on the other hand, and not surprisingly, not made for the distributed deployment. Furthermore, these systems also do not have a notion of runtime components. Thus, similarly to many of the deployment frameworks, make building autonomic systems on top of them difficult. Among the three presented systems (Java JARs, .NET assemblies and DEB/RMP Linux packages) Java JARs are the least advanced one. Namely, they do not support most of the basic package-level deployment operations, such as installation via an installation manager, uninstallation and versioning. Furthermore, standard Java environment does not provide a package repository into

which JARs can be released.

Solutions around the lack of modularity in the Java platform also, like the existing packaging systems, suffer from the lack of mechanisms for distributed deployment of code—none of them, except for the OSGI platform, is coupled with tools for the distributed deployment. Furthermore, these are mostly low-level solutions to Java-specific problems, not coupled with reflexive runtime component models. As a result, systems such as MJ, JPloy or iJAM do not provide advanced support for runtime management of components, such as dynamic updates. On the other hand, all of those systems provide extended installation-related functionality such as component repository, support for multiple versions and isolation of code. These features are mostly ignored by the deployment frameworks and models, such as JSR88, SmartFrog etc.

Based on our analysis, we believe that a key aspect of deployment of component-based applications is to properly model and manage the physical implementations of these components. This modelling is necessary because the implementations of components and the way in which they are resolved has an impact on the correctness of the runtime software architecture. Thus, the implementations need to become part of this explicit architecture. As we will illustrate in the following chapter of this thesis, such modelling facilitates the building of autonomic architecture based management frameworks for component systems.

Table 5.1: A comparative analysis of the state of the art

	Deployment frameworks					Packaging systems			Java module systems			
	DAnCE	JSR88	SmartFrog	FDF	NIX	DEB/RPM	JARs	Assemblies	OSGi	iJAM	MJ	JPloy
Support different packaging formats?	No	No	No	No	No	No	No	No	No	No	No	No
Have a notion of runtime component?	Yes	No	Yes	No	No	No	No	No	No	No	No	No
Addresses dynamic updates?	No	Yes	No	No	No	Yes	No	No	Yes	No	No	No
Interference free deployment?	No	Yes	No	No	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes
Independent deployment?	No	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes
Transparent updating?	No	No	No	No	No	No	No	No	No	No	No	No
Incremental builds?	No	No	No	Yes	Yes	No	No	No	Yes	Yes	Yes	No
Supports installation?	No	Yes	No	Yes	Yes	Yes	No	Yes	Yes	Yes	No	No
Supports multiple versions?	No	Yes	No	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes
Supports undeployment?	No	Yes	No	Yes	Yes	Yes	No	Yes	Yes	Yes	No	No
Has a repository?	No	No	No	No	Yes	Yes	No	Yes	Yes	No	Yes	No
Fine-grained import-export?	No	No	No	No	Yes	Yes	No	No	Yes	Yes	Yes	No
Supports distributed deployment?	Yes	Yes	Yes	Yes	No	No	No	No	No	No	No	No
Applicable to legacy software?	Yes	No	Yes	Yes	No	No	No	No	Yes	No	No	No

Part II

Contribution

Résumé de chaptire 6

Dans le chapitre 6 de cette thèse on explique le déploiement dans le contexte de JADE—un projet de recherche qui vise a fournir une plateforme a composants réflexives pour le développement des gestionnaires autonomes. Au début, comme la plupart d'autres systèmes de gestion base sur architecture, JADE s'est focalise sur la gestion des logiciels patrimoniaux. Par conséquence, le déploiement n'était pas considéré comme une partie intégrale de gestion des logiciels base sur l'architecture. Nos expériences montre que le déploiement doit être considéré comme une partie intégrale d'approche architectural pour la gestion des systèmes reparti. D'abord, le déploiement est une brique de base pour la plupart des opérations de gestion et reconfiguration d'architecture. Ensuite, JADE a rapidement évolué vers une conception récursive, dans laquelle JADE est lui-même construit avec les composants de JADE. Par conséquence, JADE gère JADE ce qui rends le déploiement un aspect fondamentale. Dans ce chapitre on présent comment notre système de déploiement était conu et notre solution générale dans le contexte architectural.

Chapter 6

Capturing deployment in the component-based reflexive architectures

Contents

This chapter is about considering deployment in the context of JADE, a research project that aims at providing a reflexive component-oriented framework for developing autonomic management systems. When it started, like many other architecture-based management systems, JADE (Bouchenak et al. 2005) focused on managing legacy software systems. Hence, deployment was not considered as part of the initial goals since legacy deployment tools were used. It is our experience however that deployment must be considered as an integral part of an architecture-based approach to the management of distributed systems. First, deployment underlies most of the management reconfigurations of

the architecture. Second, JADE rapidly evolved towards a recursive design where JADE is built using distributed JADE components. Consequently, JADE manages JADE and therefore deployment became the foundation of JADE. This chapter presents how our deployment was designed, its challenges, and our solution from an architecture perspective[1]

6.1 Introduction

The goal of autonomic computing (Kephart and Chess 2003) (White et al. 2004) (Ganek and Corbi 2003) is to automate the functions related to system administration. This effort is motivated by the increasing size and complexity of systems and applications alike, which has two direct consequences: the administration costs are an increasing part of the total information system costs, the difficulty of the administration tasks reach the limits of what human administrators can handle. Consequently, autonomic computing advocates self-management capabilities. It is our experience in JADE that deployment is the foundation of autonomic computing but that it has seldom been seamlessly integrated in architecture-based management systems for autonomic computing.

In JADE, we address self-management through an architecture-based approach where management is about observing and evolving the architecture of the managed systems. The role of autonomic managers is to react to observed evolutions of the managed system, re-architecturing it accordingly. Our experience is that a reflexive component-oriented approach is very effective for advanced self-management capabilities. JADE uses components to capture the traditional concept of managed elements but applies component modelling to not only managed applications but also the architecture and behavior of the underlying distributed system hosting these applications. In other words, JADE models the architecture and behavior of a complete distributed system through a component-oriented approach.

JADE advocates the use of component reflection as the foundation to support the *introspection* and *reconfiguration* of the managed distributed system. Through introspection, autonomic managers can not only observe the architecture of the managed distributed system but also its runtime behavior, including for instance its dynamic performance for detecting poor resource utilization or security-related communication patterns for detecting intrusion. Through reconfigurations, autonomic managers can manipulate the architecture of the managed distributed system, including for instance the ability to balance dynamic loads by migrating components across nodes, provide higher availability through replicating components across nodes, or changing the overall Quality of Service (QoS) by replacing certain components with others that have different QoS characteristics.

JADE advocates a minimal reflection on components that offers a uniform management interface for autonomic managers. In fact, JADE provides only a few reflexive controllers that capture the traditional management aspects such as configuration properties, bindings amongst components and lifecycle. We argue that our minimal reflexive component-oriented approach, combined with a fault-tolerant programming model, represents a suitable framework for building advanced autonomic

[1]The work around the JADE autonomic management system is a common effort of many people participating in the INRIA Rhˆone-Alpes SARDES project. I was mainly responsible for the design and development of JADE's deployment functionality.

management over distributed systems. By a fault-tolerant programming model we understand the fact that JADE applies active component-level replication as well as checkpointing of reconfiguration operations to allow for classical roll backs. Furthermore, JADE is built using JADE, following a recursive design. Autonomic managers are developed using JADE components that are distributed and replicated. Hence, JADE is itself a distributed system and JADE manages itself.

As the foundation of JADE, one finds the challenge of autonomic deployment of components in the context of a distributed system. To reach autonomic deployment, we had to take several major steps in JADE. From a traditional architecture-based approach, we first had to model the distributed system within the architecture. We also had to extend the component model to capture the implementation of components, recursively modelling component implementations with components. These two steps brings deployment within the realm of architecture-based management, into which we introduced an autonomic deployment manager.

The last step was to consider faults, which is unavoidable in distributed systems. One of our main goals in JADE was to provide a fault-tolerant environment for building autonomic systems. The approach was through an autonomic self-repair manager that detects failures and repairs them by reconfiguring the architecture to repair what has been lost in the failure. The self-repair manager is also able to self repair itself, since the self-repair manager is itself a set of distributed and cooperative JADE components. For the self-repair manager to function properly, it needs not only to rely on a deployment manager, but it needs that the deployment manager be fault-tolerant, which impacted our early design of our deployment manager.

This chapter is organized as follows. In Section 6.2, we introduce the architecture-based design of JADE. In Section 6.3, we introduce modules that capture the concept of component implementations, extending the JADE architecture. In Section 6.4, we capture deployment in the architecture-based paradigm, extending the JADE architecture again and introducing the autonomic deployment manager. In Section 6.5, we present two case studies that illustrate deployment in JADE. In Section 6.6, we conclude.

6.2 JADE Management System

In JADE, we address self-management through an architecture-based approach where we model the architecture and behavior of a complete distributed system through a component-oriented approach. In particular, JADE also advocates the use of component reflection as the foundation to support the *introspection* and *reconfiguration* of the managed distributed system. Through introspection, autonomic managers can not only observe the architecture of the managed distributed system but also its runtime behavior. Through reconfiguration, autonomic managers can manipulate the architecture of the managed distributed system. JADE advocates a minimal reflection on components that offers a uniform management interface for autonomic managers. We argue that our minimal reflexive component-oriented approach, combined with a fault-tolerant programming model, represents a suitable framework for building advanced autonomic management over distributed systems.

6.2.1 Component model

The component model underlying the JADE autonomic management system follows the definition of a software component given by Clemens Szyperski (Szyperski 1998) —

> *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

Accordingly, any managed element is a component in JADE. More formally, JADE adopted the FRACTAL component model (Bruneton et al. 2004) — a general model which is not tied to a specific programming language or software technology. Through Fractal, JADE can cleanly separate the *content* and *control* aspects of components. The control aspect is about the management control of components provided by a set of control interfaces, more on this below in Section 6.2.2.

The content aspect of a FRACTAL component carries the functionality of that component, following a service-oriented paradigm. A component provides services, through *server interfaces*, and requires services, through *client interfaces*. Through interfaces, FRACTAL advocates a strong encapsulation for better software engineering, offering an effective mean for hiding the implementation details of components.

This explicit definition of requirements is one of the important differences between components and objects. Explicit requirements make assembling components by third parties possible. In such component assemblies, component dependencies are satisfied through *binding*. FRACTAL supports both primitive and composite bindings. Primitive bindings directly connect a required client interface to a provided server interface. In contrast, composite bindings are built out of chained primitive bindings, allowing to build complex communication schemes such as remote stubs or dynamic service adapters.

6.2.2 Membrane model

The membrane reifies the control aspect of a component, providing the foundation for the JADE management operations. In JADE, each functional component is associated with the following five controllers:

- Attribute controller
- Interface controller
- Binding controller
- Lifecycle controller
- Containment controller

Each controller supports both the introspection and reconfiguration of the metadata it reifies. The attribute controller allows to observe and change the set of configurable properties understood by the component content. A Java signature of this controller is illustrated in figure 6.1. The *listFcAtt* method returns a table containing the names of the attributes. The *getAttribute* method takes the name of an

attribute as a parameter and returns its value. Finally, the *setAttribute* method takes a name and a value of an attribute as parameters and assigns the given value to the given attribute.

```
public interface AttributeController {

  public String[] listFcAtt();
  public Object getAttribute(String name);
  public void setAttribute(String name, Object value);

}
```

Figure 6.1: The Java signature of the *AttributeController* interface

The interface controller (also called the *Component* control interface) allows to navigate the client and server interfaces of a component. Each of the interfaces of a component has a name. Thus, as illustrated in the Java signature of this controller interface presented in figure 6.2, the *getFcInterface* method takes a name of an interface as an argument. It returns a value of this interface, which is either an implementation of an interface if the interface is of a server type, or a reference to the component's required service if an interface is of client type. The *getFcInterfaces* method returns a table containing all the component's external interfaces – the client, the server and the control ones.

```
public interface Component {

  Object getFcInterface(String interfaceName)
  Object[] getFcInterfaces()

}
```

Figure 6.2: The Java signature of the *Component* interface

The binding controller allows to observe and modify bindings between components Any component which has a client interface must also have a binding controller. The Java signature of this control interface is illustrated in Figure 6.3. The *bindFc* method takes two arguments: a name of a client interface belonging to this component and a reference to a server interface to which the given client interface is to be bound. The *listFc* method returns all the names of client interfaces that this component has. The *lookupFc* method takes a name of a client interface and returns a reference of the server interface to which this client interface is bound. Finally, the *unbindFc* method unbinds a given client interface from its associated server interface.

The lifecycle controller allows to observe and change the running state of its component, such as started or stopped. This controller provides a set of methods to perform the life cycle operations, as illustrated in figure 6.4 The *startFc* and *stopFc* methods allow for a transition between two possible states of the component lifecycle: *STARTED* and *STOPPED*. The current life cycle state of a component can be introspected by calling the *getFcState* method, which returns one of the two lifecycle states.

The containment controller allows to observe and change the hierarchical containment of components, as described in the next section below. The Java signature of this controller's interface is illustrated in figure 6.5. The *addFcSubComponent* method takes a reference to a component as an

```
public interface BindingController {

  public void bindFc(String clientItfName, Object serverItf);
  public String[] listFc();
  public Object lookupFc(String clientItfName);
  public void unbindFc(String clientItfName);

}
```

Figure 6.3: The Java signature of the *BindingController* interface

```
public interface LifeCycleController {

    public static final String STARTED;
    public static final String STOPPED;

    public void startFc();
    public void stopFc();
    public String getFcState();

}
```

Figure 6.4: The Java signature of the *LifeCycleController* interface

argument and adds this component as a child of the component to which this controller belongs. The *getFcSubComponents* method returns a table with references to all the children of the component to which this controller belongs. Finally, the *removeFcSubComponent* method takes a reference to a component as an argument and removes this component from the list of children.

```
public interface ContentController {

  public void addFcSubComponent(Component subComponent)
  public Component[] getFcSubComponents()
  public void removeFcSubComponent(Component subComponent)

}
```

Figure 6.5: The Java signature of the *ContentController* interface

Each controller is easy to understand, providing only a few methods, but all five controllers provide powerful low-level mechanisms that enable full introspection and reconfiguration of complex architectures. Moreover, FRACTAL controllers constitute a uniform management interface. This means that most management functions become generic in that they only manipulate functional components through controllers.

6.2.3 Containment model

JADE leverages the hierarchical nature of the Fractal component model to provide a better structuring of the architecture of the managed system. The Fractal model distinguishes between *primitive* and

composite components. Primitive components only encapsulate functionality, such as business logic, providing server interfaces. Composite components only encapsulate a group of components, either primitive or composite. Hence, composite components provide a hierarchical structure to an otherwise flat assembly of components. It is important to point out that components may be shared between one or more composite components.

Figure 6.6 illustrates a sample assembly of FRACTAL components. The client interfaces are represented in green, the server ones in red. Within the primitive components one can see their implementations.

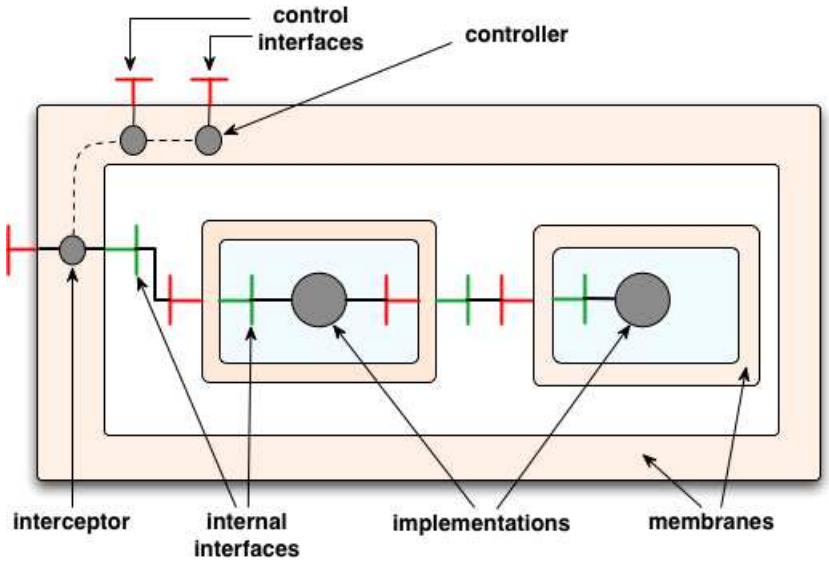


Figure 6.6: An internal content of a fractal component

Composite components have several important roles in JADE. Hierarchical composites are important from the architectural point of view in that they allow to define the level of abstraction at which the system's architecture is introspected by the management software. Furthermore, composites have an important role in managing the life cycle, binding and sharing of their subcomponents.

Composites also define binding scopes. When not shared, components can only be bound between siblings, that is sub-components of the same composite. In other words, a component that is not shared must have all its bindings satisfied by its sibling components. When shared, a component can have its bindings satisfied by its sibling components across several composites.

The use of shared components can prove very useful in the context of dynamic assemblies of components, like the one illustrated in figure 6.7, a *Client* component can be dynamically deployed outside of the *Composite A*, yet it needs to be bound to the *Server* which is the subcomponent of the *Composite A*.

In this example, one possible solution is to expose the server interface of the *Server* component as a server interface of the *composite A*. However, this is not always possible as it requires the change in the set of interfaces exposed by composite A. Another solution uses sharing. A new composite component, called the *composite B* is created. The *Client* and *Server* components become the children of this *composite B* with two consequences. First, the *Server* component becomes shared between

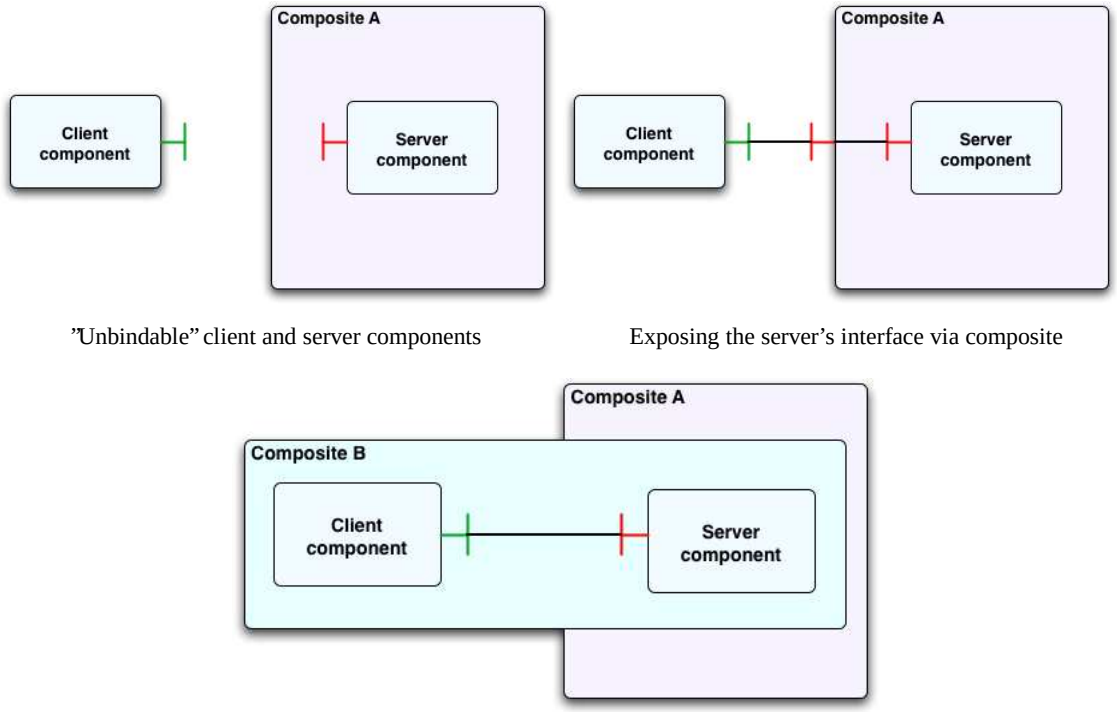


Figure 6.7: An example usage of sharing

composite A and *composite B*. Second, both components *Client* and *Server* can now be bound together as siblings within the *composite B*.

Composites also define a *group lifecycle* for all their sub-components. When a composite is started, it also starts its subcomponents. The order in which subcomponents are started is defined by the order in which the subcomponents were added to the composite. When the composite is stopped, it recursively stops all its subcomponents, in the same order.

6.2.4 Factories and deployment

The creation of components in JADE happens through *component factories*. It is customary, but not mandatory, that there is a single factory per running environment. A factory is defined as a functional interface, illustrated in Figure 6.8.

```
public interface Factory {

    public Component newFcInstance(Type type, Object controllerDesc, Object contentDesc);

}
```

Figure 6.8: The Java signature of the *Factory* interface

To create a new component, one calls the *newFcInstance* method on a factory. The first argument

of this method is the type of the component to create. The type of a component defines the contract between components of that type and the outside world. In the FRACTAL model, a type is the set of client and server interfaces that represents what a component requires from surrounding components and what it provides to these components.

The type does not imply an implementation. Making a parallel with object-oriented languages, a component's type is similar to an interface of an object, specifying a behavior. It differs from a language interface type in that a type not only specifies the behavior of the component (the server interfaces) but also the requirements of that component (the client interfaces). A component type also differs from a language class that combines a behavior and an implementation (both a structure and method bodies).

Therefore, to complete the type information, the factory needs some information about the implementation. In fact, this information is two-fold because a component is made of a content and a membrane. The membrane is described through a domain specific language that essentially tells what implementation to use for each controller, following a mixin approach for an efficient code generation of the membrane. The content description is relatively simple in the case of composite components since FRACTAL composites have a generic implementation as one that groups sub-components.

For primitive components, a factory needs something that will represent the implementation of the functional content of the component to create. In Java for instance, this information is the name of the implementation class for that component. In the C language, it would be an entry point in a shared library as it is the case in binary-level component models such as COM or XPCOM. In COM for instance, a component is created through a CLASSID that is an 128-bit world-global and unique identifier of an implementation, itself provided as a Dynamically Loadable Library (DLL).

When creating a new component, the factory returns the *Component* interface of that component. Any FRACTAL component must implement this server interface that uniquely identifies that component throughout its lifetime. By default, there are no other means to identify or locate components within a reflexive architecture. For instance, components do not necessarily have unique and immutable names and they do not have any other form of binary identification.

6.2.5 Reflexive Architecture

The *reflexive architecture* in JADE captures the architecture of the assembly of components. It is the foundation of the architecture-based approach advocated by JADE for system management. The reflexive architecture is a meta-level description leveraging the presence of membranes in the FRACTAL component model. The five controllers described previously provide the core reflection necessary for management operations, allowing not only to introspect the reflexive architecture but also supporting reconfigurations of that architecture.

In other words, JADE supports *full runtime reflection* that combines the ability to both *introspect* and *reconfigure* the reflexive architecture. This runtime reflection supports the programming model for JADE managers that are autonomic control loops. An autonomic manager observes all or part of the reflexive architecture and maintains some invariants such as a certain quality of service or repairs failures or smooths out workload variations. The autonomic goal is that no human intervention be necessary in the dynamic maintenance of these invariants. Human intervention should be kept to tuning these invariants and the decision making process of the corresponding policies to achieve them.

Composites play an important role in the reflexive architecture provided by JADE as they supply a *view mechanism* on the architecture itself. Using a uniform approach, a hierarchy of composites captures a view of the architecture such as a hierarchy of functional features of the system, a hierarchical clustering of components on the different nodes of a cluster, or simply a group of replicas as JADE supports the transparent replication of components.

Views therefore facilitate the communication of high-level ideas and design decisions about various aspects of the reflexive architecture, similarly to what blueprints do in the context of building architecture. Through views, managers can introspect and reconfigure the reflexive architecture from the one perspective provided by the view they are using, such as a functional bindings, component placement on nodes, or component replication. As such, views demonstrate the importance of component sharing between composites. For example, a *logger* component can be a subcomponent of an application, which gives one view of it as a functional part of an application, and at the same time, the same *logger* component would be a subcomponent of a node, which gives another view of it as being deployed on that node.

Because JADE is advocated to build self-managing distributed systems, it is important to consider failures and their impacts on the programming model JADE provides. To resist component failures, JADE maintains a checkpoint of the reflexive architecture, called the *System Representation* (SR). Indeed, when a component fails, both its content and membrane fail. This is perfectly illustrated by process failures, where the entire implementation of a component is lost, which includes both its membrane and its content. In such events, JADE looses the parts of the reflexive architecture that were captured by the lost membranes.

This loss is jeopardizing the ability to build autonomic self-repair systems, that is, systems that autonomously repair their own failures. Indeed, a self-repair manager would need to know what was the reflexive architecture prior to a detected failure in order to rebuild what was lost. This knowledge was exactly the one that the lost membranes captured. Note that the original scripts in the FRACTAL ADL (Architecture Description Language) are useless here since JADE supports designing dynamic systems that most probably evolve after their initial creation.

Therefore, the checkpoint dynamically captures a consistent system representation. JADE controllers are responsible to maintain this System Representation up to date as a side-effect of committing reconfigurations of the reflexive architecture. Indeed, autonomic managers express their reconfigurations as atomic reconfigurations.

6.2.6 Autonomic managers

As mentioned previously, autonomic managers are control loops that observe and reconfigure the running system through the JADE reflexive architecture. Each manager typically observes a view of the architecture and when necessary plans and executes a reconfiguration. The observation usually relies on the presence of *probes* that reify both reflexive and hardware characteristics such as failure detectors, performance monitors or component-specific feedback.

The reconfiguration relies on the concept of a reconfiguration plan. A plan is a sequence of operations on the architecture, expressed as reified method invocations on the membranes of certain components. For example, one may wish to add a sub-component to a composite, which is represented in the plan by the reification of an invocation to the method of the containment controller. This

reification includes the identity of the receiver of the future method invocation (the composite) as well as the argument values (the sub-component to add).

The plan is a runtime data structure that fully leverages the reification of the reflexive architecture as well as the underlying reflective capabilities of Java, the programming language used to develop managers in JADE. First, all component entities in the plan are identified using their reification in the SR, as first-class components. As we mentioned earlier, components do not have any special means of identification other than the reference to their component interface. For JADE managers written in Java, the identity of a component simply appears as a Java reference to an object implementing the language interface *Component*.

Second, the reification of the plan requires the use of Java reflection in order to reify the method invocations of the plan. Through reflection, Java supports building runtime data structures that describe full method invocations, including the Java method to invoke, the future receiver of this invocation and the future arguments to use in that invocation. Beyond the other usual reasons for choosing Java as a programming language, the reflective capabilities of Java were a major rationale.

6.3 Capturing Modules

The early versions of JADE carried over the traditional approach to component-oriented programming, which proved a strong limitation in the context of building autonomic management systems.

Traditionally, component-oriented programming is about a service-oriented paradigm where components provide functional services that need to be composed into a consistent assembly. The focus is on the mechanisms needed to express and enact this assembly. Most component-oriented approaches do limit themselves to this early phase, usually adopting a descriptive approach based on an Architecture Description Language. An assembly is expressed in the ADL and an ADL factory processes such ADL description in order to build an assembly of components that can be started.

JADE adopts a more dynamic approach to component-oriented programming, reifying at runtime the reflexive architecture for managers to observe and reconfigure. The starting point is the same however, JADE uses an ADL description of an initial assembly that is processed by a specific ADL factory, creating an initial assembly of components and therefore the corresponding reflexive architecture. Autonomic managers can then use this reflexive architecture to not only observe and report about the architecture but also reconfigure it when necessary. The focus remains however on the functional aspects of the component assembly.

The forgotten aspect is the management of the implementations of components. Many component-oriented frameworks consider the management of the implementations of the components they manage as something external to component-oriented programming. It is assumed that an underlying platform takes care of implementation concerns. A perfect example is the use of Java to support component-oriented programming. In most approaches, the implementations of components are JAR files that need to be made available in the class path of the running Java Runtime Environment (JRE). This approach works in many cases, but proved insufficient for JADE and our goal to support autonomic management system in distributed environments.

The stumbling stone is that implementations do carry specific dependencies that also need to be captured in the reflexive architecture. Failure to do so simply prohibits autonomic managers to have

a complete view of the architecture and therefore potentially prevents them from planning correct reconfigurations. Let's illustrate this in JULIA (*Julia: Fractal Composition Framework Reference Implementation, Objectweb* 2002), the Java incarnation of FRACTAL, since Java is an environment most people are familiar with.

Using JULIA, one can create components and describe an initial assembly. Let's further assume this assembly is consistent and can actually be built and started. At the end of this process, a runtime instance of the JADE environment is up and running, hosting a set of components. Some time later, a new version of a component is published fixing some important bugs, which suggests to update the component used in our running system with the newly available version. The types of both versions of the component are verified to be compatible, in other words, the new version of the component provides the same server interfaces and requires the same client interfaces. All seems perfect, the new component can safely replace the old one, the update reconfiguration can take place.

A reconfiguration plan is generated. The first step is to determine which components need to be stopped. This starts with the component to be updated. Other components bound to that component also need to be stopped; this is of course a recursive concern and the entire transitive closure of components bound to components that need to be stopped must be stopped. Then, the plan must include the installation of the new version of the component and the removal of the old version. Then, bindings can be reconstructed so as to rebuild the assembly using the new version of the component. Once bindings are in place, the stopped components can be restarted. The expectation is that everything will start smoothly.

The reality can be shockingly different. The new version of the component may be using some native library that fail on the version of the underlying operating system. Without being that crude, the new version of the component may fail because it expects a newer version of the Java Runtime Environment. For instance, it uses the W3C XML DOM API and assumed it was available from the JRE, which is only true after Java version 5.0. Even more subtle, the new version of the component may provide the same component type, with the same server and client interfaces, but they may use slightly different versions of the corresponding language interfaces. Indeed, FRACTAL does not capture such information about component interfaces. It uses the name of language interface but does not capture any versioning information about such interfaces.

Capturing modules is about solving this problem. We capture the implementation of modules in the architecture and we do so using components. Thereby, implementations are not only visible to managers but the dependencies of these implementations are also visible. We describe this extension in the following subsections.

6.3.1 Extending the component model

We extended the component model of FRACTAL to include the concept that every component has an implementation, modelled as components. We call the implementation of a component a *module*. Therefore, every component has a module. A module is itself a component, which introduces a classical recursion, as in object-oriented programming languages with reflection. An object is an instance of a class, itself an object and therefore instance of a class. The recursion stops with a meta-class, instance of itself. Our recursion stops with a meta-module, module of itself.

The binding concept in FRACTAL is generic enough to capture the dependency between a com-

ponent and its module, but such a binding reflects a meta-level dependency. Most incarnations of component models approach bindings as functional dependencies. These functional dependencies are usually supported by services, that is, objects implementing language interfaces. Although we usually perceive these concepts as applied to a business logic of components, they are quite general and equally apply to a meta-level.

The binding between a component and its module is indeed a meta-level dependency. It does not reify a functional dependency in most cases[2]. It reifies a runtime dependency between the component and its implementation, as needed by the execution engine. In C, this is the dependency between the running component and the shared library loaded to provide that component with the code and static data it needs. In Java, this is the dependency between the component and the class loader providing the classes that component needs.

In Java, modules are usually represented with class loaders, which is a functional component from a meta-level perspective. A class loader reifies classes and provides a scope for Java types that is used by the execution engine. At runtime, the Java virtual machine will invoke the class loader to obtain the classes needed by the running component. The class loader has a functional interface, it is invoked as a service; but it is not explicitly invoked by the functional code of components, except for components that explicitly manipulate Java runtime reflection.

This binding between a component and its module is however special regarding the component's lifecycle. The component cannot be instantiated before its module is available, meaning created and started. Indeed, without an implementation, a component may not be created. This is different than a regular binding where the component is created first and its binding are created later. For regular bindings, the lack of a binding may prevent the component from being started but not from being created.

Symmetrically, the loss of the module binding requires the component to be disposed of. This is an interesting side-effect of considering modules as components. As such, modules may be affected by reconfigurations of the architecture. They may have to be stopped, as any regular functional component might be. However, when stopping a module, all the components that use that module must not only be stopped, as it is the case when revoking a regular binding, but they also need to be disposed of.

6.3.2 Module Resolver

The *Module Resolver* establishes bindings between module components, something that requires special care as module bindings are more complex and more flexible than most other bindings. We are not discussing here the binding between a component and its module, which obviously needs to be based on the identity of the module. We are discussing here the dependencies between modules. These dependencies exist because modules are components themselves.

Regular bindings are often considered as rather simple and inflexible. In many component-oriented approaches, bindings are using names that identify components. This is inflexible in that such bindings do not express what is needed – they express who provides what is needed. In FRACTAL, bindings are left unspecified in the model and can therefore use any technology and be as flexible

[2]For Java and other reflection-aware programming languages, this binding may also be used functionally

as one needs.

However, in most incarnations of FRACTAL, names are used and rely on composites for controlling the visibility scopes of such names. The names however do not name components but interfaces, introducing an interesting degree of flexibility: a client interface is bound to a server interface if the two interfaces have correct names and their two components are siblings within a composite.

For module dependencies, we need to revisit how dependencies are expressed. First, a simple name is no longer sufficient. Implementation dependencies strongly suggests to introduce versioning of implementation artefacts as implementations do evolve to fix bugs or improve functionality. The versioning semantics must capture this evolution and express compatibility rules between versions. We decided to use the same version semantics as the OSGI specification, a versioning scheme already in use for years in successful OSGI-based middleware platforms such as Eclipse, JOnAS, or IBM WebSphere.

The versioning model is as follows. A version is composed of four tokens: major, minor, micro, and qualifier. When two versions have different major tokens, the versioned artefacts are allowed to be incompatible. They don't have to be, they might be. It is extremely important for practical reasons to allow implementation artefacts to break backward compatibility, even though all efforts must be made to maintain backward compatibility as long as possible. The other tokens express evolutions that are expected to be backward compatible, such as internal bug fixes or extensions to the public behavior of the versioned artefact.

The second aspect to discuss about module dependencies is their granularity. Some approaches argue for the importance of module-level dependencies, such as Eclipse and bundle-level requirements. We feel this is an unfortunate decision in most cases as such dependencies are dependencies, tying together what is needed and who provides it. In particular, this approach does not support simple reorganizations such as splitting a module in one or more smaller modules, something quite natural to maintain the reflexive engineering quality of a module whose implementation grows.

We adopted a smaller granularity, expecting modules to have exported services like any regular FRACTAL component. At a model level, the exact semantics of such exported services cannot be specified; such semantics is incarnation dependent. Indeed, the exact semantics may differ for different incarnations of FRACTAL. For instance, Java incarnations are likely to use a granularity of Java type packages. An incarnation of FRACTAL in the C language may use a similar granularity but corresponding to the definition of a header file.

Whatever the granularity is, module dependencies can be modelled as regular FRACTAL bindings and perceived as meta-level dependencies. In both our previous examples, in Java and C, the exported services are indeed used at runtime to link the various component implementations together. We will name such services *module services*. A module service is a FRACTAL server interface that is named and versioned. An import is a FRACTAL client interface that specifies not only the name of the needed server interface but also its version. In practice, we need to consider a range of compatible versions, potentially covering versions with different majors.

Given how imports and exports are expressed for module services, we need to discuss how they are resolved, that is, how we create bindings amongst them. Regular bindings are scoped by composites, following a hierarchical approach. The resolution is straightforward since both imports and exports are expressed as simple names. Beyond the obvious impacts of considering versions, the more

important point to discuss is the adequation of hierachical resolution approach of FRACTAL, based on composites. One could wonder if this hierarchical approach is still adequate for resolving module dependencies.

The answer can be approached in two steps. First, the presence of composites does not imply any granularity on these composites. Hence, one could consider that all modules are sub-components of one unique composite, thereby creating a flat shared scope for resolving module imports and exports. In many situations, this approach is perfectly acceptable, as demonstrated by most existing incarnations of component-oriented systems that do not use runtime scoping of implementations. For instance, Julia, the Java incarnation of FRACTAL, does not use class loaders to isolate the various implementations of components.

Furthermore, in some implementations, the use of a single composite to group all modules may even be mandatory if there is no possibility to achieve runtime scoping of implementations. An incarnation in the C language using an unmodified dynamic linker and C compiler would be challenged to support multiple implementations of modules.

The second step remains at the model level, abstracting the potential runtime limitations of various incarnations. Conceptually, from a binding perspective, there is no difference between components and modules. Hence, composites are equally applicable to modules and provide a powerful structuring principle. Of course, modules will be shared for the same reasons regular components are shared.

6.3.3 Considering versions

Introducing versions for module services goes much further than just adding version meta-data to both client and server interfaces of modules. It is actually one crucial point of design that could favor or hinder the establishment of successful component ecosystems.

The danger goes as follows. Different modules may export different module services but may also export the same module service at the same version level or at a different version level. Hence, the situation may be that the same version of a module service may be exported by several distinct modules as well as different versions of the same module service be exported by different modules. Depending on how one resolves the various imports of modules, one may produce an assembly that is consistent or not.

The consistency of an assembly needs to be considered at two levels. First, we consider modules and if they are resolved or not. A module is said to be resolved when it has all its imports resolved (bound) to exports. This is known and visible in the reflexive architecture as modules are regular components and their bindings are reified as any binding between components are. The second level concerns the functional components. Even though modules are resolved, depending on how versions are handled, the assembly may still be inconsistent regarding functional components.

To illustrate clearly this complex situation, it is easier to consider the practical case of Java. In Java, a module translates into a class loader, a low-level mechanism for scoping Java types, that supports the encapsulation of component implementations. Our prototype as well as OSGI platforms consider module services at the granularity of Java packages. We could have the following situation: two functional components are bound through a regular service implementing the Java interface named *org.foo.Foo*. This interface uses a Java type named *org.bar.Bar* as part of the signature of some of its methods (either as parameter types or return types, it does not matter).

This settings requires that each component has a module with two distinct imports — one for the *org.foo* package and the other for the *org.bar* package. Each import expresses also a version constraint. Let us assume that all imports express the same constraints as an interval of acceptable versions such as $[1.0.0; 2.0.0[$ that specifies a compatibility with the major version one. If we further assume that different versions of both packages are available, we can discuss consistency depending on how imports are resolved on exports.

The correct choice is that all imports are resolved to the same exports. In this case, both functional components see the same package *org.foo* and *org.bar* and their execution will be flawless. If we assume that they see the same *org.foo* package, but different versions of the *org.bar* package; the service binding between functional modules could be considered valid, given the available meta-description. The client interface is bound to a server interface with the proper name and implementing the same language interface (*org.foo.Foo*). However, this is an inconsistent configuration because of the different resolution of the *org.bar* package. At runtime, when the two functional components will interact, passing instances of the *org.bar.Bar* types, runtime cast exceptions are more than likely to happen.

It is also interesting to point out that bindings regarding functional services are impacted by considering modules and multiple versions. Indeed, we can see that a binding for a functional service between two components is only correct if both components have the same visibility of the language type that the service implements. Please refer to Chapter 8, Section 8.1 for full details on the problem in Java.

For our design, we retained the simpler mono-version approach. We can still have different modules exporting module services with different versions, nothing is changed there. However, the resolver that creates the bindings between modules will only retain one unique version of each exported module service. The resolution process thereby ensures that all modules importing the same service, with potentially different version ranges, will see the same exported service. The problem described above disappears and the resolution process is greatly simplified.

The drawback of this design is that certain configurations could only resolve partially where it could be argued that they should fully resolve. Indeed, some assemblies could have different modules needing different versions that are all available (exported by other modules). There are situations where it would be correct to use these different versions and other where it would be incorrect (as illustrated in our example above). It all depends on the degree and nature of the integration of functional components. It is interesting to consider the impacts of advocating a support for multiple versions or not on the overall component ecosystem.

Assuming that one supports multiple versions, we believe that one favors an ecosystem that is turned towards backward compatibility and therefore slower to adopt evolutions. This choice has an advantage though — it is able to resolve assemblies with different modules needing different versions, even potentially incompatible ones. As explained above, this only works if functional components whose modules are resolved with different versions do not interoperate (or at least not around the software artefacts resolved differently). Therefore, the validity of the approach is related to the degree of interoperability between applications. With traditional applications, mostly standalone, the support of multiple versions is a major win. For instance, two standalone applications could be resolved with different versions of the same toolkit library for their graphical widgets.

With a more middleware–like approach, where applications tend to cooperate at a finer granularity, chances are that supporting multiple versions will impair the actual runtime cooperation while the corresponding assemblies are considered resolved and correct from a dependency perspective. Consistent with the position of statically typed languages, we decided to err on the side of safety. Our design considers only one version, which is justifiable only if managed components are not only assembled for standalone operations but actually cooperate. This suggests an interesting use of composites to isolate different parts of the overall systems, allowing the best of both worlds. However, the challenge will become the detection of resolution problems and deciding which hierarchy of composites avoids them.

It is interesting to further discuss the different heuristics that can be used in resolving modules, within the design choice of keeping a single version for each exported artefact. The major heuristic concerns the choice of the kept version when several are available. Choosing the older versions seems hardly arguable. However, a choice exists between the most recent version and the version that resolves the most importers.

Locally, it seems that this latter choice is the most favorable, and it is. However, it is interesting to discuss the expected effects on the ecosystem at large. If the ecosystem is alive and thriving, the tendency will be for newer versions to resolve many importers. The burden will be for providers of older versions of components and their modules to release newer versions, staying on top of the evolutions of the functionality they depend on.

If the ecosystem is slow to evolve, the tendency will be that older versions will satisfy more importers on average. The consequence is that newer components will have a difficult time being installed and resolved in most runtime systems. This indirectly will limit the ability for novelty to succeed and will most probably lead to the death of the ecosystem, although nothing is certain since component technology is rather new and that evolution of cooperating assembly has neither been a reality nor a subject of study.

It is interesting to note that the needs of one particular system are somewhat contradictory with the survival needs of the overall ecosystem, not something unlike the similar conflicts in natural ecosystems between the immediate needs of the individuals and the longer term needs of the community.

Given a system that needs to be reconfigured, the tendency will be to favor preserving already installed components and therefore choose older versions. Indeed, already installed components are likely to be used by end users to manage important data. Therefore, end users would rather see a system preserving the already installed and running software and their corresponding data. But this in turns would make it harder for newer software to develop a market, something essential for the longer-term viability of the ecosystem.

In our design, we favored the choice of the version that satisfy the most importers. It is a practical choice that is justified to preserve the maximum usability of the running system that is being reconfigured. We expect that human intervention will be necessary to tweak this behavior. For instance, it must be possible for an end user to choose to favor newer components at the price of loosing older ones. This can be easily manipulated by end users through expressing the relative importance of components or higher-level features composed of components. With such hints, the resolver can then take care of the details of finding which older components will be invalidated. A feedback loop approach could be conceived where the configuration management and the end user (or administrator)

will iterate several times before striking the right balance.

6.3.4 Module API

Since from the architectural point of view modules are regular components, their manipulation is performed exactly like manipulation of any other types of components within the application. This means that modules are created and managed using the same API as any other components. The specific aspects of module components are defined in their implementations as well as in their bindings. For example, a Java-based module component would have an implementation that encapsulates a Java class loader and a binding that defines a delegation scheme between class loaders.

Before a module can be instantiated, its type needs to be defined. In FRACTAL, like in many other component models, a type of a component is defined by this component's set of client and server interfaces, as previously described in Section 6.2.4. In case of module components, the names of those interfaces are important in that they define the inter-module dependencies processed by the resolver. Thus, a name of every interface contains information on the provided or required resources, as well as the versions of these resources.

The resolution of modules is implementation-specific – it can either be triggered automatically, every time a new module is created within the local repository, or it can be forced. In the former case, the module repository awaits notifications of module creation from the component factory or, if the repository is at the same time the factory of module components it knows when new modules are added to the system. In case when the module resolution is forced, an external manager calls the resolver's *resolve* method, which launches the process of module resolution.

Once modules are resolved, functional components can be created from them. For that, the factory interface previously described and illustrated in figure 6.8 has been extended. The extended version of this interface provides several ways of specifying which module is to be used for a given functional component.

```
public interface Factory {

   public Component newFcInstance(Type type, Object controllerDesc,
        Object contentDesc, IModule module);

   public Component newFcInstance(Type type, Object controllerDesc,
        Object contentDesc, Component moduleComponent);

   public Component newFcInstance(Type type, Object controllerDesc,
        Object contentDesc, Object moduleComponentDesc);

}
```

Figure 6.9: The Java signature of the extended *Factory* interface

Each functional component created using the extended component factory keeps an immutable reference to this factory. Every such component also exposes an extended life cycle controller interface (see Section 6.2.2 for the description of the basic life cycle controller). The extended life cycle controller provides the *undeploy* method. When this method is called on a stopped and unbound functional component, the factory which created this component is notified of the component's

destruction. The factory then, in turn, notifies the garbage collector component of the component destruction. This allows the garbage collector to remove a potentially unused module component from which the given functional component was created, as well as the unused dependencies of the given module component. The Java signature of the notification interface is illustrated in figure 6.10.

```
public interface IGarbageCollector {

        public void notifyComponentCreated(Component comp);

        public void notifyComponentDestroyed(Component comp);

}
```

Figure 6.10: The Java signature of the garbage collector interface

6.4 Capturing Deployment

Capturing deployment in JADE is about the core mechanisms necessary to support the deployment of components in the context of a distributed system. In fact, deployment capabilities are required by almost all architecture-based management operations and thereby constitute the foundation of a framework like JADE.

Given the reflexive architecture described in the previous section, capturing deployment has several dimensions. The first one is deciding how deployment will be surfaced to developers of autonomic managers in JADE. From day one, we wanted to re-use our architecture-based approach, modelling deployment as architecture reconfigurations. This suggests capturing the underlying distributed system as part of the overall reflexive architecture, using components.

The second dimension is to model physical packages that are the concrete artefacts that need to be copied onto a given node in order to be able to instantiate a component. In the previous section, we introduced modules as the implementation of a functional component. Modules reify the loaded implementations while physical packages are the artefacts on disk. A typical example is the difference between a loaded and linked shared library and its corresponding image on disk. Physical packages are locally managed at nodes in a local repository.

The third dimension is the concept of a reconfiguration plan as the collaboration medium between autonomic managers. Autonomic managers do manipulate the reflexive architecture, producing plans for reconfigurations. Most managers will produce plans that are abstract from any deployment details. At a certain level, the distributed architecture is virtualized, that is managers see virtual nodes. They produce plans expressing constraints on these virtual nodes, the deployment carries out these plans translating virtual nodes into real nodes taking into consideration the physical constraints of deployment. For instance, a component may have an implementation only for Linux systems and not for Windows.

The deployment manager also augments the plan with module-related considerations. All existing JADE managers are focused on the assembly of functional components. The role of the deployment manager is to complement any reconfiguration plan about functional components with the necessary

architecture reconfiguration for creating and binding the necessary modules needed by these functional components. Of course, this includes the deployment of the modules themselves on nodes where they are needed by functional components.

In this regard, the deployment manager has an essential responsibility: scheduling the execution of the various operations for reconfiguring the architecture. For instance, a physical package has to be deployed onto a node prior to creating a module from it. Another example is that a functional component can only be created once its module has been created from a corresponding physical package, resolved, and started.

This section is structured as follows. We present the different dimensions of component deployment in the first subsections. In Section 6.4.1, we discuss the modelling of a distributed system in JADE. In Section 6.4.2, we discuss physical packages and their local management. In Section 6.4.3, we detail the concept of reconfiguration plans and the different responsibilities of the deployment manager. In Section 6.5, we illustrate component deployment with two important use cases. The first one presents a deployment for the GRID and the second one illustrates the cooperation between the autonomic self-repair manager and component deployment.

6.4.1 Modelling distributed systems

Modelling distributed systems in JADE goes much further than just modelling nodes and their network connections. Of course, it is about extending the reflexive architecture with a virtual model of a distributed system. This allows to approach deployment through architecture reconfiguration, keeping the JADE programming model consistent. But more fundamentally, it is about recognizing that JADE itself is a distributed system composed of both managed and managing components that are physically distributed on networked nodes. This is the main rationale behind our recursive design where JADE itself is built using FRACTAL components, opening up the possibility of JADE managing JADE. In particular, this means that JADE can deploy the components of itself.

Recursive design

The foundation of a recursive design is the separation of a base layer and its meta-level layer. For the recursive design of JADE, the meta-level is a set of *manager components* that manage a set of *managed components* that constitute the base layer. Of course, as it is always the case with recursive designs, manager components are also managed components. Therefore, any component may manage others through their controllers but any component may also be managed by other components through its own controllers.

Hence, the first autonomic capability to achieve in JADE is deployment, which suggests to reify the underlying distributed system as managed components. To achieve this, JADE introduces the concept of JADE nodes, illustrated in figure 6.11. A JADE node represents a physical node of the underlying distributed system. The main goal of JADE nodes is to serve as factories of software components to the deployment manager running within the JADE boot. JADE nodes are therefore composite components into which deployed components are added as sub-components. This applies for both modules and functional components.

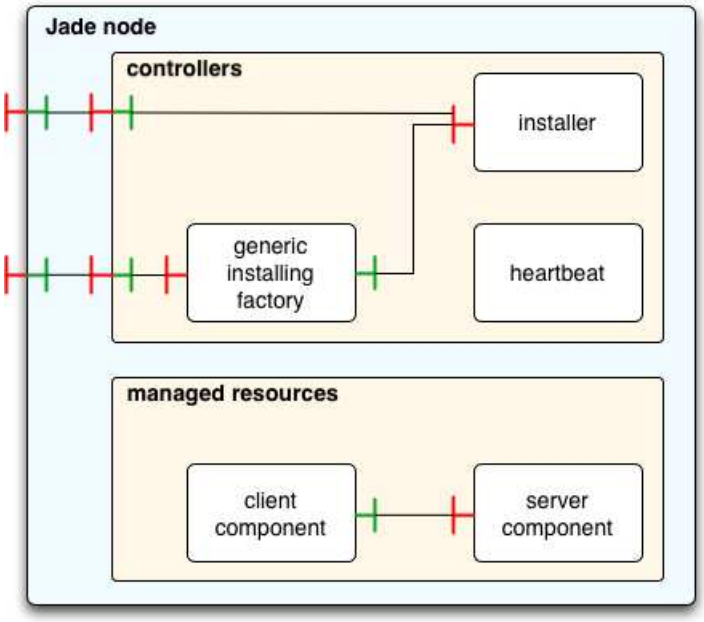


Figure 6.11: JADE Node

This very first autonomic capability illustrates very well the originality of JADE's design: the architecture not only captures software elements but also hardware elements such as physical machines. JADE also reifies other system information such as main memory, processing power or stable storage capacity. Other system information are more dynamic such as monitoring probes that provide runtime feedback on the load of nodes. All this is modelled using sub-components to the composite representing a physical machine.

The use of components keeps the programming model consistent. A manager that wishes to use remote monitoring probes will do so using the reflexive architecture. It will model its probes as components and will deploy them by declaring them as subcomponents to the composites reifying the nodes where its probes need to be deployed. It will use these probes through regular bindings, expressing a dependency as any component would do on another component. Such bindings will be composite bindings however, transparently handling the remote nature of the probe location. Another example would be about failure detection using heartbeats. Each node will have a heartbeat component that generates heart beats. Such heart beats are multicasted to a group of components on other machines that will be able to detect the machine failure if heart beats are no longer received. These communications are also expressed as regular bindings between components.

Generalizing, any management function in JADE is implemented using components, deployed over the underlying distributed system. Hence, components and their controllers are a distributed system whose architecture is self-describing. A manager is therefore a component interacting with this distributed system, evolving its architecture.

Fault-tolerance

[3]

Beyond deployment which we have just discussed, the distributed nature of JADE gives raise to fault-tolerance concerns, especially the single-point of failure that any manager represents, or any component for that matter. To address this concern, JADE advocates a fault-tolerant programming model.

Fault-tolerance is introduced through active replication at a component level. The choice of an active replication is natural for transparently replicating components; indeed, most components are naturally developed as deterministic. Furthermore, an active replication does not require any extra development effort from component developers as there is no need for a component-specific protocol to propagate updates between replicas.

The introduction of a transparent replication capability within JADE fully leverages its recursive and reflexive component-oriented design. While JADE developers perceive non-replicated components being assembled through unicast bindings, JADE runtime has to handle the corresponding multicast and gathercast semantics to and from replicated components. An incoming binding to a replicated component has to be intercepted and transparently translated to a *multicast semantics* onto the different replicas. An outgoing binding from a replicated component has to be also intercepted and transparently translated to a *gathercast semantics*, ensuring that any architectural manipulation requested by the different replicas is executed only once. This is achieved transparently through composite bindings.

Replication provides a way to protect components from partial or temporary failures of the underlying distributed system, such as node failures or network failures. Replication is however not enough in our context – it has to be combined with the overall atomicity of architectural reconfigurations, evolving the architecture from one consistent state to another. Indeed, architectural reconfigurations may be rejected by controllers for various reasons such as security policies or lack of appropriate connecting technologies between legacy systems.

This overall atomicity is transparently provided by the JADE runtime. All JADE controllers offer architectural manipulations that can be undone. For example, a bind operation has the unbind operation as a reverse operation. Another example is adding a sub component to a composite that can be later removed. Consequently, JADE can automatically provide atomicity for architectural reconfigurations using a classical roll-back mechanism that apply reverse operations.

This scheme has to be extended to functional cooperation between managers. Indeed, managers may not be limited to manipulating the FRACTAL controllers of the components they manage; they may themselves be developed as cooperating meta-level components. For instance, most managers do interoperate with the deployment manager. Atomicity has to be preserved also regarding this functional cooperation. This requires managers to offer reverse operations as well.

6.4.2 Introducing physical packages

[3] The work around fault-tolerance is mainly the work of Sylvain Sicard, as his Ph.D. thesis. It is presented here to provide the necessary background to understand the cooperation between the self-repair and deployment managers.

A **physical package** is a physical container that groups a set of resources, both code and data, such as compiled code or image files. From a physical package, available locally on disk, the JADE runtime knows how to instantiate a module. As a concrete example, the physical package can represent a JAR file if a component model is implemented in Java or an ELF file for a component implemented in the C programming language.

The management of physical packages represents the first layer of deployment. Each JADE node provides such a management, being able to receive physical packages from the network and store them in a local repository. It is essential that this information be part of the reflexive architecture, modelled as any other architecture information. Hence, each physical package is modelled as a component. When available on a JADE node, the component reifying a physical package appears as a sub-component of the component reifying that JADE node.

This uniform approach allows JADE managers to observe what physical packages are available on which JADE nodes. In particular, any one wanting to deploy a physical package P to a JADE node N can do so simply by adding the component reifying the physical package P as a sub-component to the composite component reifying the JADE node N. Internally, the JADE system will transfer a copy of that physical package P to the JADE node N. To resist failures, physical packages can be replicated across JADE nodes. The internal transfer will therefore find a live replica, close to the target node, and initiate the copy from that replica to the target node.

Also because of failures and to simplify deployment, each JADE node can garbage collect its own local repository of physical packages. Each JADE node tracks and removes unused physical package components from its local repository. A physical package is not used when no local module is currently instantiated from that physical package. Each JADE node has access to the necessary information through local component factories that create and remove components, including modules.

6.4.3 Reconfiguration plans

Any manager that wishes to reconfigure the reflexive architecture does so through a deployment plan that it passes to the deployment manager. A reconfiguration plan is a set of core operations on specific JADE controllers. A plan can be as simple or as complex as necessary. It can be the deployment of entire GRID application over five thousand nodes around the world or it can be as simple as changing bindings between two components.

When a plan is passed to the deployment manager, it is a raw sequence of architectural reconfiguration operations, reifying future invocations on the controller methods of certain JADE components in the reflexive architecture. In most cases, this plan focuses on the functional reconfiguration of managed components and does not include any of the necessary reconfiguration regarding the deployment of functional components on the various nodes of the underlying distributed system.

This section presents the various transformation that the deployment manager performs on the raw plan in order to achieve the reconfiguration that it represents. We first introduce the plan itself. Next we discuss the deployment specific transforms on that plan, producing a complete plan containing all the necessary management operations to carry out the desired reconfiguration of the architecture. Finally we discuss the distributed execution of the complete plan.

Raw plan

A plan is a set of ordered *Tasks*. Each task encapsulates an operation performed on the basic control API exposed by JADE controllers (see Section 6.2.2). The plan is written in a domain specific language. This language allows to encapsulate the control operations within the tasks as well as to express dependencies between these tasks. There are essentially three types of tasks within the domain specific language:

- The *Regular* tasks
- The *Require* tasks
- The *Provider* tasks

The regular tasks, when executed, simply perform the control operations on the software components – they do not return a result. Like any other type of tasks, they can depend on other tasks in which case the other tasks are always executed first. The require tasks provide the possibility to specify a reference to a task on which they depend. This way, the require tasks can use the result of the execution of a task which they require. Finally, the provider tasks when executed return a result which can be used by the tasks that require them.

For example, the *Component Creation* task reifies the future invocation of a JADE factory to create a new component. It is a *provider* task in that the execution of this task returns an instance of a newly created software component. The *Binding* task reifies the future invocation of *bind* method of the binding controller on a given component in order to establish a binding with another component. It is a *Require* task in that it requires a component creation task to provide it with a reference to two components between which the execution of this task will establish a binding. These are only a few examples of the tasks within the raw reconfiguration plan and of the dependencies between these tasks. The plan uses tasks which cover all the controller functionality provided by the component model underlying JADE.

It is important to note that the raw plan only uses tasks which concern functional components. This plan does not contain information on physical nodes into which these functional components are deployed, or on the modules which the components require in order to be instantiated.

Completing a plan

Every autonomic manager in JADE is capable of producing a deployment plan. Most autonomic managers will however be focused on the assembly of functional components. It is therefore the responsibility of the deployment manager to complete the produced plan with the missing information.

Raw plans are expressed using virtual nodes that need to be mapped onto available physical nodes. Virtual nodes allow autonomic managers to express hints about which components need to be co-located on the same machine and which components need to be placed on different machines. The deployment manager will choose the actual nodes of the underlying distributed system based on several criteria that are part of the internal policy of the deployment manager. As any autonomic manager, the deployment manager can be replaced or modified in the JADE framework.

Our default deployment manager is designed for cluster environments and therefore uses the load of nodes as the essential heuristic. However, we do not impose all the nodes of the cluster to be identical machines, with identical hardware and identical operating systems. Therefore, when choosing a cluster node, our deployment manager takes into consideration if it has an implementation available for the hosting environment (operating system and hardware).

Hence, choosing real nodes is done in conjunction with extending the plan with the necessary architectural reconfiguration steps regarding both modules and physical packages. As regular components, modules and packages have to be deployed as the other functional components in the raw plan. Of course, modules and physical packages are co-located with the functional components that depend on them.

At the end of this phase, the deployment plan contains all the necessary reconfiguration operations to carry the entire reconfiguration, including the deployment specific operations regarding the containment manipulation of the composites reifying the target nodes. This includes adding new components (functional, modules, and physical packages) to these composites as well as removing the unnecessary ones.

Execution of a plan

One essential responsibility of the deployment manager is to execute the complete plan, distributing that execution on the various target nodes and locally scheduling the various tasks in the correct order.

Although the reflexive architecture appears as centralized to an autonomic manager when introspecting it, the reflexive architecture is a distributed system. Hence, a plan that reconfigures the architecture is a distributed process. In our prototype, the plan is interpreted locally and the deployment manager relies on a remote method invocation substrate to invoke methods on remote controllers. This approach has the advantage of simplicity for the one that developed the deployment manager but also from a fault-tolerance perspective. The plan succeeds or fails, leaving no orphans distributed on concerned nodes.

It would be interesting to pursue this work and experiment with distributing the execution of the plan, shipping sub-parts of the plan to concerned nodes for local execution, similarly to what is described in (Quéma et al. 2004) – an approach in which modules are not taken under consideration. The plan partitioning seems simple at first since we know all the receivers of the controller method invocations that need to be done and we know where these receivers are located on physical nodes. However, many operations will be cross-node operations such as creating bindings between remote components or starting a composite whose sub-components are distributed on different nodes. An examample scenario could be a composite representing a multi-tiered web system, such as the complete Java EE solution. In this scenario, each tier is a subcomponent of the Java EE solution composite. The tiers are often located on different network nodes, mainly for performance and administration reasons. These tiers need to be started in an orderly manner, i.e. the database should be started before the business tier and the business tier before the presentation tier. Furthermore, each tier can be duplicated within a composite representing a clustered solution. In this “view”, each composite encapsulating duplicated tiers, which are also normally located on different network nodes for fault-tolerance reasons, should start its children in a parallel manner, to optimize the tier’s start-up time.

This clearly shows that the efficient distributed execution of the complete reconfiguration plan is

a complex subject that needs further study. This work has focused on the other dimension of this execution: the ordering of the various tasks in the plan. Indeed, tasks carry implicit synchronizations as some depend on the results of others tasks. The following list gives an idea of the main scheduling dependencies between tasks:

- A component can only be created once its module has successfully started.
- A binding can only be done on a created component, not yet started.
- Starting a component can only happens once all the necessary bindings are available
- A module can only be instantiated on a JADE node if its corresponding physical package is available on that node.
- A component must be stopped prior to be disposed of.
- A component must be disposed of before its module is stopped.

Ordering the tasks is not sufficient. For the reconfiguration plan to be valid, all modules on all concerned JADE nodes must resolve. This requires that the deployment manager runs a resolver on modules, per concerned JADE node. The resolver solves the constraint problem represented by the required client interfaces and the provided server interfaces amongst modules. The result of this resolution step is a set of bindings between modules and the final decision about which modules are resolved, that is, have all their required client interfaces bound to an acceptable exported server interface.

The resolver usually attempts an incremental approach, avoiding to unbind any pre-existing bindings so to limit the disruption of the execution of the reconfiguration plan. But sometimes it is best to destroy some of the existing bindings to allow for a more consistent overall configuration of modules. How aggressive a deployment manager is in reconfiguring already running components will be an interesting trade-off to study as dynamic component-oriented programming will become mainstream.

The resolution phase could be distributed, as the overall execution of the plan. However, it can also be done locally as long as it is divided with respect to nodes. In other words, the resolution process uses the node containment view in the reflexive architecture. Through this view, the deployment manager can introspect the current location of components, including modules, and therefore produce one constraint system to be solved per target node, for the modules deployed on that node.

Once we had computed the bindings between modules and that we have the knowledge of those that are resolved and those that are not, the execution of the complete plan can continue. All tasks are organized in three phases: all the stop operations on local components that need to be stopped. All the necessary bind and unbind operations on stopped components. For components loosing their modules at this stage (unbind operation on the binding to their module), they have to be disposed of. Finally, all the start operations on stopped components that can be started (the resolved ones).

If anything goes wrong, the reconfiguration is considered to have failed and it is rolled-back. It is interesting to point out that the deployment of physical packages and module components does not need to be rolled-back as each JADE node has a local garbage collector for the asynchronous clean-up of local repositories of these packages and modules. It is however essential that all other operations be undone so to bring the overall architecture back to a consistent state.

6.4.4 Plan implementation details

The notion of the reconfiguration plan described above is a general one. In the existing prototypes of a deployment systen for JADE described in chapters 7 and 8 we have implemented several ways of expressing and executing the architectural reconfiguration tasks contained within the plan.

The first prototype of a deployment system for JADE, described in chapter 7, is essentially an extended FRACTALADL factory. In this respect, this work is similar to the ones described in (Abdellatif et al. 2005) and in (Flissi and Merle 2006). The deployment system therefore reuses the ObjectWeb task scheduling framework. This framework is a Java API for defining a set of tasks, which can be executed in an orderly manner. Based on this API, we have extended the set of tasks with the ones responsible for allocation of physical nodes, installation of physical packages and instantiation and resolution of modules. This prototype however is limited in terms of cooperation between autonomic managers. Moreover, the tasks describing architectural reconfigurations are essentially about building an initial architecture, not modifying an existing one.

In the second prototype of JADE, described in chapter 8, the reconfiguration plan is also expressed in terms of a Java API. This work has been realized by Sylvain Sicard, as part of his PhD thesis. Compared to the work described in chapter 7, the plan is externalized from the ADL factory and can be exchanged between the autonomic managers. In both prototypes of JADE the plan is expressed in terms of Java code, which essentially encapsulates future calls on the component factories and controllers provided by the JADE framework.

Further investigation of possible implementations of the deployment plan are beyond the scope of this thesis. As illustrated by the work of Christophe Taton in his PhD thesis, one possible approach is to use a high-level, concurrency- oriented language, such as Oz, to provide a *dynamic adl*. Thanks to its built-in parallelism and synchronization features, Oz allows to express complex workflows in a natural manner. The work of Christophe proves that this allows languages like Oz to be used succesfully for describing architectural reconfigurations of component-based software. Such reconfiguration descriptions are “externalized”, thus can be exchanged between autonomic managers or serialized. They are also more compact and powerful in terms of their expressiveness than similar programs written in the Java language.

6.5 Case studies

This section presents two essential categories of deployment scenarios that JADE handles. The *initial deployment* is an ADL-based deployment that assumes an empty system beyond the core JADE platform. It assumes that a software has to be deployed on a known set of target machines where an empty JADE runtime is already present. This is a common case in the context of GRID computing.

The *incremental deployment* corresponds to a dynamic reconfiguration of an already running JADE system. It usually is a result of incremental activities performed by autonomic or human managers that wish to reconfigure the reflexive architecture. We will use the self-repair case as it demonstrates the full gamut of dynamic reconfigurations of the reflexive architecture.

6.5.1 GRID-like deployment

Deployment in the context similar to the one of grid computing (*Grid Computing Info Centre* 2002) is a specific case of software deployment. For one, grid-like environments are built using a large number of heterogeneous machines. For two, each deployment within such context is an *initial* one since grid applications are standalone applications only sharing the underlying grid infrastructure. Namely, when an application is deployed on the grid it is given a *clean* set of *empty* machines on which no other software components execute. Thanks to the virtualization, these machines may not be real machines, but this is irrelevant here. Once the distributed application is deployed on the grid, it does not evolve until the user decides to free the resources and perform another *initial deployment*.

In this context, JADE provides an Architecture Description Language (ADL) to describe initial assemblies of components. The ADL is XML-based and initial configurations are provided via XML files. An ADL description of an initial assembly contains information about the software components to be deployed, the nodes on which each component is supposed to run, and how components are to be interconnected (bindings). By default Jade's ADL files do not contain information on modules and physical packages, as illustrated in figure 6.12

```
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
  "classpath://org/objectweb/fractal/adl/xml/basic.dtd">

<definition name="examples.ClientServer">
  <component name="Client">
    <interface name="c" role="client" signature="Service"/>
    <content class="ClientImpl"/>
    <virtual-node name="node1" />
  </component>
  <component name="Server">
    <interface name="s" role="server" signature="Service"/>
    <content class="ServerImpl"/>
    <attributes signature="ServiceAttributes">
      <attribute name="header" value="->"/>
      <attribute name="count" value="1"/>
    </attributes>
    <virtual-node name="node1" />
  </component>
  <binding client="client.c" server="server.s"/>
</definition>
```

Figure 6.12: An example JADE architecture description file

In the above ADL description, a user of the grid decides to deploy a simple client-server application which consists of two functional components: the *client* and the *server*. The client component has a client (required) interface named *c*, has an implementation class *ClientImpl* and is to be deployed on a node symbolically named *node1*. The server component provides a server interface called *s*, which has the same signature as the client's interface *c*, has an implementation class *ServerImpl*, is to be deployed on a *node1*, and has two configurable attributes. Finally, the user specifies that client's required interface is to be bound to the server's provided interface.

Given such a description, the ADL-based deployment engine of JADE parses it and performs preliminary verifications such as checking that all the mandatory component interfaces have corre-

sponding bindings etc. Once the verification is successful, the deployment engine creates a raw deployment plan, as described in Section 6.4.3. In the initial version of JADE, this raw plan was almost the complete plan as modules and physical packages were not modelled as components. The actual deployment of software artefacts was simply performed under the hood with ad-hoc deployment tools and infrastructures.

In the current version of JADE, the raw plan has to be completed with modules and physical packages. As illustrated in Figure 6.13, the information about modules can be included in the ADL description. If not, the deployment manager would use available meta-data about known relationships between components and modules.

```
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
  "classpath://org/objectweb/fractal/adl/xml/deployment.dtd">

<definition name="examples.ClientServer">
  <component name="Client">
    <interface name="c" role="client" signature="interfaces.Service"/>
    <content class="ClientImpl"/>
    <virtual-node name="node1"/>
    <module name="Client" version="1.0.0"/>
  </component>
  <component name="Server">
    <interface name="s" role="server" signature="interfaces.Service"/>
    <content class="ServerImpl"/>
    <attributes signature="ServiceAttributes">
      <attribute name="header" value="->"/>
      <attribute name="count" value="1"/>
    </attributes>
    <virtual-node name="node1"/>
    <module name="Server" version="1.0.0"/>
  </component>
  <binding client="client.c" server="server.s"/>
</definition>
```

Figure 6.13: An example JADE architecture description file with modules

In the above figure the *Client* component's implementation is contained within the *ClientModule-1.0.0* module, whereas the *Server* component's implementation comes from the *ServerModule-1.0.0* module. The modules are described in a separate architecture description file for the purpose of clarity. An example module description file is illustrated in Figure 6.14.

We can see that each module is associated with at least one physical package. Furthermore, we can see that this file describes the dependencies between modules. The syntactic differences should not be understood as perceiving modules as different than regular components, they are not. Modules are regular components and their imports and exports represent regular FRACTAL interfaces that will have regular FRACTAL bindings.

The *Client* module imports the *interfaces* package from some other module and comes with a set of resources contained within the *client.pkg* physical package. The *Server* module on the other hand exports (provides) the *interfaces* package in version 1.0.0 and also comes with a set of resources, contained in the *server.pkg* physical package.

Once the ADL-based deployment engine has generated the complete deployment plan, using both

```
<modules>

  <module name="Client" version="1.0.0">
        <import package="interfaces" version="1.0.0"/>
        <import package="system" version="1.0.0"/>
        <content file="client.pkg"/>
  </module>

  <module name="Server" version="1.0.0">
        <export package="interfaces" version="1.0.0"/>
        <import package="system" version="1.0.0"/>
        <content file="server.pkg"/>
  </module>

</modules>
```

Figure 6.14: A sample module description file

description files illustrated in Figure 6.13 and Figure 6.14, it executes the plan, scheduling appropriately the different tasks. This starts with the installation of physical packages *client.pkg* and *server.pkg* on virtual node *node1*. Follows the creation of the module components for the *Client* and *Server* modules, binding them together, and starting them. The execution ends with the creation of application components from the correct modules, binding them, and starting them.

6.5.2 The self-repair case

Self-repair is certainly one of our most advanced autonomic managers in JADE. It ensures not only the autonomic self-repair behavior of the components it manages but it is also capable of self-repairing itself. As such, the self-repair provides the foundation of the JADE programming model for developing advanced management capabilities that are fault-tolerant. The autonomic self-repair manager relies heavily on the presence and fault-tolerance of the autonomic deployment manager for its proper function. This case study perfectly illustrates our *incremental deployment* scenario.

Architecture-based self-repair

Our self-repair advocates an architectural recovery process that, after a failure, re-establishes a valid reflexive architecture of the managed system. Our current failure model for nodes is a fail-stop one. A fail-stop model says that a node may fail but it assumes that the node was working correctly until it stopped. This fail-stop model ensures that no failed manager has performed erroneous reconfigurations of the architecture. This assumption, combined with the overall atomicity property of architectural reconfigurations, maintains the overall consistency of the reflexive architecture.

Our self-repair is an entirely generic process at the architecture level. The capacity to self-repair does not imply any specific pattern on the reflexive architecture of the managed application. It does not require any specific capability from managed components beyond the JADE controllers.

Once a failure is detected, the self-repair manager reverts the managed system to an earlier consistent state that existed prior to the failure detection. This roll-back approach is composed of the following three main steps: fault detection, fault analysis, and fault recovery.

The detection step. This step detects fail-stop faults appearing in the system. It monitors the health of the managed system through probes installed on nodes; these probes use a heartbeat technique to detect node failures. In other words, nodes are self-watching for failures.

The analysis step. This step analyses the detected fault by introspecting the reflexive architecture in order to understand how to repair the detected fault.

This step only works if the reflexive architecture is fault-tolerant. Indeed, when a node fails, all components running on that node also fail, including their controllers. This means that JADE looses some parts of its reflexive architecture. Without proper measures, this would mean that the self-repair manager has lost the very information it needs to know what needs to be repaired.

It is important to recall that JADE is a totally dynamic system where the architecture evolves at run-time. Only the reflexive architecture reflects the true state of the managed system. Consequently, our design ensures that the reflexive architecture is tolerant to node failures. This is done by introducing a checkpoint capability that checkpoints the last known consistent sate of the reflexive architecture.

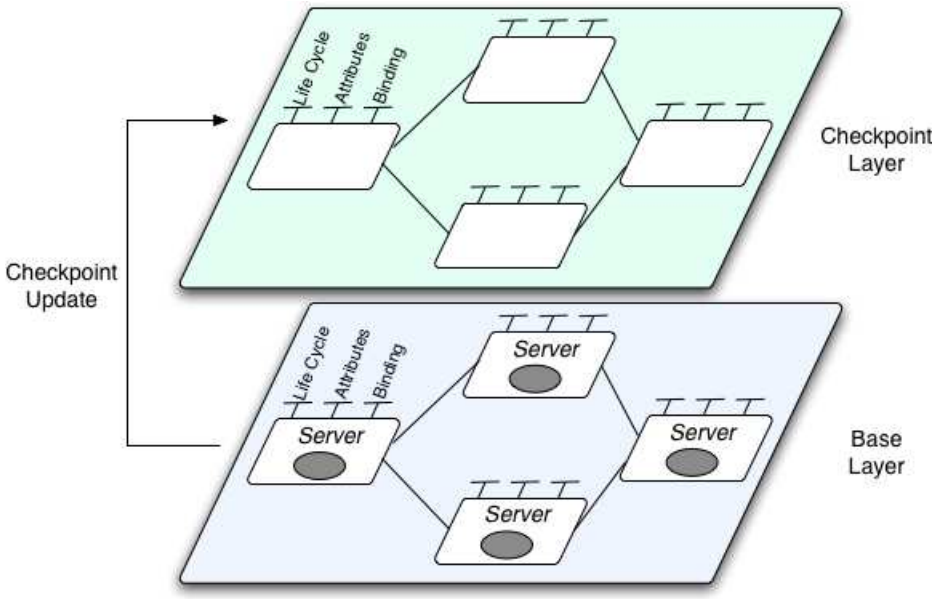


Figure 6.15: Management and Checkpoint layer

The checkpointed architecture appears as regular components, as depicted in Figure 6.15. It is the responsibility of JADE controllers to maintain the checkpoint of the architecture as a side effect of commits of architectural changes requested by autonomic managers. The rationale for designing the checkpoint using components is to maintain a uniform reflexive interface for any manager that needs to access the architecture, be it the reflexive architecture or its checkpointed counter part.

The repair step. The checkpointed architecture restores the ability of the self-repair manager to observe a consistent architecture once a failure has been detected. It can therefore build a plan to repair what has been lost. During this repair step, the self-repair manager allocates a new available

node, uses the deployment service to deploy the components that need to be instantiated on the new node, re-establishes the bindings and containment (composites) for these newly created components and finally restarts components that were running prior to the failure.

This scheme works well for stateless components, the self-repair manager only needs to restore the architecture, simply re-creating components that were lost in the failure. This is perfectly applicable for the various tiers of a J2EE Web Application Server, as long as the underlying file system is accessible from the various nodes of the cluster. For the database component, it will usually require to restore the failed node since database systems usually require using local disks. If stateful components wrap legacy systems that offer checkpointing capabilities, these can be used. For pure JADE components, the use of persistence frameworks such as JDO or Hibernate could be envisioned.

Recursive self-repair

Self-repair suggests a recursive design that exploits the overall recursive design of JADE, based on components. Indeed, the self-repair manager watches over managed components and repairs them when a failure is detected; but the self-repair manager itself needs to be watched and repaired in case a failure affects its operation. We term this the self-self-repair.

For self-self-repair, we need to add fault-tolerance to both the self-repair process itself and the Checkpoint layer that is used by the self-repair process. This almost comes for free in JADE if care is taken to consistently apply the fundamentals of architecture-based management using JADE reflexive and component-oriented framework. Because both the Checkpoint layer and the self-repair manager are both developed as JADE components, we can leverage JADE's ability to replicate components and provide a fault-tolerant self-repair service.

By being a replicated component, the self-repair manager can self-repair itself. First, the self-repair manager detects failures of replicas of itself. Second, it is able to re-create failed replicas, like any other component lost due to a failure. In doing so, the self-repair manager maintains the cardinality of its own replication, without human intervention. The transactional semantics of the JADE programming model ensures that failures during repair attempts do not compromise the reflexive architecture.

Being a replicated component, the Checkpoint layer is guaranteed to resist node failures. We also know that the checkpoint layer only captures consistent architecture state since JADE controllers atomically update the checkpointed architecture, as we already mentioned.

By designing the Checkpoint layer with components, we introduce a recursion that needs to be controlled. Indeed, as the checkpoint layer represents the architecture of managed components and since it is itself composed of components, the checkpoint layer contains a representation of its own architecture. Since changes in the reflexive architecture create components and therefore controllers, the very architecture of the Checkpoint itself changes since it mirrors the reflexive architecture in terms of components. This recursion is illustrated in Figure 6.16. To avoid this problematic recursion, the controllers of the components used to implement the Checkpoint layer are specialized in that they do not attempt to reflect their own architecture in the Checkpoint layer.

The operating principle to gain self-self-repair capability can be summarized as follows. First, there is one self-repair manager that uses one Checkpoint layer. Second, both are implemented as JADE components that are transparently replicated by JADE. Replication provides a first step towards

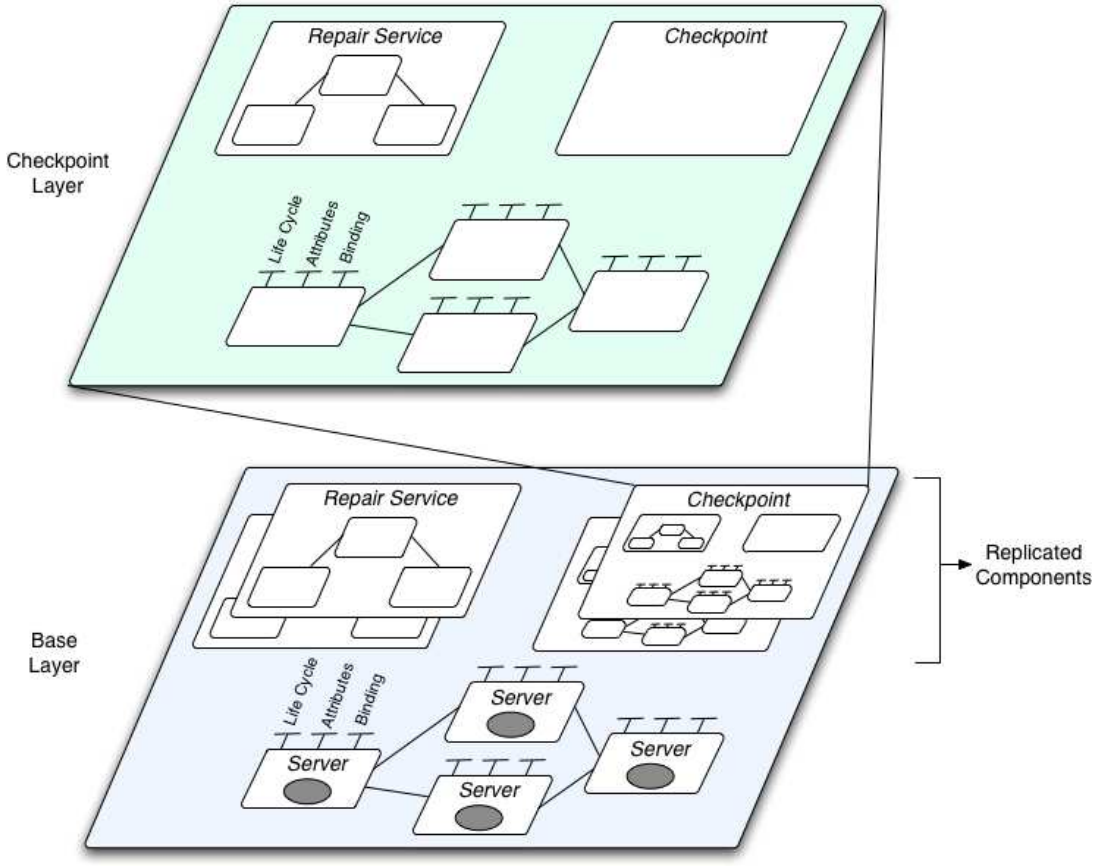


Figure 6.16: Putting pieces together: repair service & replicated components

self-self-repair; the second step is provided through the ability of the self-repair manager to maintain the replication cardinality. Third and final step, the consistency of the reflexive architecture is ensured by the atomicity of architecture reconfigurations.

Relationship with deployment

The self-repair manager relies on an autonomic and fault-tolerant deployment manager for its proper function. Like any other manager, the self-repair manager relies on the deployment manager to complete its raw reconfiguration plans. However, it also relies on the deployment manager for part of the fault-tolerance of its resolution of the recursion induced by the self-self-repair property.

Regarding the completion of raw reconfiguration plans, the deployment manager has the same responsibility as explained in Section 6.4.3. In this regard, deployment supports the fault-tolerance requirement of JADE. First, through its ability to choose physical nodes and deploy components on them, it supports the replication strategy for providing fault-tolerance to autonomic managers. Second, it supports the ability of the self-repair manager to rebuild what has been lost in a failure through re-deploying components on new nodes and rebuilding bindings.

The design of our deployment has been also impacted in more subtle ways. By replicating the local

storage of physical packages on several nodes, the deployment participates in fault-tolerance since it avoids loosing some component implementations in case of node failures. Moreover, the presence of garbage collection at each local node simplifies the implementation of the atomicity of architectural reconfigurations. Indeed, there is no need to explicitly remove modules or physical packages – unused ones will be discovered asynchronously and removed locally, without a global decision. However, for fault-tolerance reasons, the positive decision on removal of a package is taken only if there are enough available replicas of the given package on other nodes. This will be explained in detail in section 8.5 Of course, any such removal will be reflected atomically in the reflexive architecture.

A more important point is the replication of the deployment manager itself to provide the necessary fault-tolerance to the self-repair manager in order to solve the recursion introduced by its self-self-repair ability. As part of repairing itself, the self-repair manager depends on its ability to maintain the cardinality of its own replication. To do this, it relies on the deployment manager as it needs to re-deploy a new replica of itself on a new node. By replicating the deployment manager, we ensure enough fault-tolerance to node failure to ensure the proper functioning of our overall fault-tolerance scheme.

The actual design of the self-repair manager is the subject of the Ph.D. thesis of Sylvain Sicard, in the Sardes team at INRIA. We have worked together in the interaction between the two goals of autonomic self-repair and self-deployment. Sylvain's work includes the replication scheme and its FRACTAL mechanisms, I am responsible for the design of the deployment manager and in particular the design of the local repository of packages and the module layer.

6.6 Conclusion

In this chapter, we captured deployment as an integral part of the architecture-based approach to system management. We did this work in JADE, a research project that aims at providing a reflexive component-oriented framework for developing autonomic management systems. Through the years, deployment has imposed itself as the foundation of JADE, underlying most if not all management operations through architectural reconfigurations. However, the journey has been long and treacherous, as the next chapter illustrates through the various attempts at introducing deployment in JADE.

One of the main reasons for this was that JADE evolved from a traditional approach to architecture-based system management to an advanced framework for building autonomic systems. In the traditional approach, the focus was on managing legacy systems, independently deployed through legacy deployment tools and infrastructures. This puts deployment as a necessary evil, but not captured by the architecture-based paradigm. Through the years, JADE evolved towards a fundamentally novel approach: a framework and a distributed platform for building autonomic systems. The approach is still compatible with managing legacy systems, but it uses a recursive design where JADE itself is developed as a distributed component-oriented systems.

Hence, JADE manages JADE, putting deployment at the foundation of the JADE architecture and a central design issue. In particular, JADE provides a fault-tolerant distributed paradigm for developing autonomic managers that rely on both the self-deployment and self-repair capabilities of JADE. This required not only a cooperation between deployment and self-repair, but it strongly suggested to consider a uniform approach to modelling the architecture of the managed system: JADE itself. From a

deployment perspective, we extended the component-oriented model for capturing the implementation of modules as well as the modelling of a distributed system, bringing deployment from the status of an external and necessary evil to the status of an autonomic manager, providing many of the core bootstrap functionalities for other autonomic managers to work properly.

Part III

Implementation

Résumé de chaptire 7

Dans le chapitre 7 de cette thèse nous décrivons la première version de système de déploiement pour JADE. Cette version est basé sur deux technologies existants—le modèle a composants FRACTAL et la plateforme a service OSGI (voir 7.1). On a adopté OSGI car cette plateforme fourni une solution avancée aux problèmes de modularité de l'environnement Java. Cette aspect de OSGI complète bien le aspects dynamiques fourni par JULIA—une incarnation Java de modèle FRACTAL.

Dans la section 7.1 de ce chapitre, on décrit OSGI et JULIA pour donner une base des connaissances nécessaires pour la compréhension de la reste de ce chapitre. Section 7.2 illustre comment cette version de JADE gère les logiciels patrimoniaux. Ensuite, dans la section 7.3 on présente l'architecture reparti de JADE, en se focalisant sur le déploiement et sur la représentation système. Puis, dans la section 7.4 on décrit la console d'administration de jade, qui offre les capacités avances de déploiement pour les administrateurs humains. Dans la section 7.5 on évalue cette première version de jade, avant de conclure ce chapitre dans la section 7.6.

Chapter 7

JADE: First Design

Contents

This is the original design of JADE. It is an architecture-based approach to the management of complex and distributed legacy systems. This a two-layer design: the JADE system and the managed systems. The approach is architecture-based and component-based such as to model managed systems as components. The component-based approach provides a uniform management APIs, the controllers.

In its early years, JADE targeted the autonomic management of complex and distributed legacy systems. By autonomic, we mean that JADE could manage the legacy systems without, or with minimal, human intervention. The approach was architecture-based which means that the architecture of the managed legacy systems is reified at runtime for autonomic managers to observe and reconfigure if need be.

One crucial and early design point was to adopt a reflexive component-oriented model as the foundation of JADE. We chose the FRACTAL model that not only provided components but also reified the assembly of components. In FRACTAL, a component is both a content (its functional part) and a membrane (its control part). FRACTAL does not impose any particular set of controllers but JADE imposes the set of five controllers detailed in Section 6.2.2. These five controllers provide the core management operations that are basis of autonomic managers. In other words, autonomic managers can both observe and reconfigure the architecture through these five controllers.

Through components, the goal was to model the architecture of complex legacy systems and prove that the approach advocated by JADE made it simpler to write autonomic control loops. Three early autonomic managers were initially considered: self-protection, self-repair, and self-optimization. All three were written using the FRACTAL controller API, a task that proved much simpler in JADE than

in many other management systems. The manipulation of a high-level modelling of the architecture through only the five concepts of the FRACTAL membrane proved highly effective.

Another main reason for this software engineering efficiency was that JADE managed to hide most of the distribution challenges. The first challenge regarding distribution is that the managed legacy systems were distributed. This meant that managers had to observe and reconfigure a distributed architecture. The second challenge was that the distributed nature of the legacy systems suggested that a deployment solution had to be provided—deployment underlies almost all architecture-based reconfigurations.

The two challenges found a unique solution in a distributed design of the JADE platform. We decided to combine several existing technologies. We adopted Java for its large acceptance and its portability. We adopted the OSGI platform for its advanced support for dynamic modularity; something that perfectly complemented the JULIA technology, a Java highly-optimized incarnation of the FRACTAL model.

This chapter presents this first design of the JADE based on these various software technologies. In Section 7.1, we cover the necessary background on OSGI and JULIA technologies. In Section 7.2, we present how wrappers are built for legacy systems. In Section 7.3, we present the overall distributed architecture of JADE, focusing on deployment and the system representation of the architecture. In Section 7.4, we discuss the administration console that offers advanced deployment capabilities for human administrators. In Section 7.5, we evaluate this first design of JADE in the context of the success story of Jasmine. In Section 7.6, we conclude.

7.1 Background

This section is a background section about the OSGI and JULIA that underlay JADE. The OSGI technology is used for its advanced support for modularity while the JULIA technology is used for its advanced support for components, following the reflexive FRACTAL model.

7.1.1 About OSGI

The OSGI Technology is a specification from the OSGI Alliance that defines a Java platform for the dynamic assembly of network deployed components. The design target was long-running system where software components could be dynamically added and removed, without having to shut-down and restart the OSGI platform.

Several implementations of the OSGI specification exists and some have been the foundation of very successful projects, both open source software and proprietary products. For example, the Eclipse platform is based on OSGI, using a home-brewed implementation called *Equinox*. On the server side, *Equinox* is used as the platform of the IBM WebSphere Application Server. Similarly, in the ObjectWeb open source community, the JONAS Web Application Server is also based on OSGI, using the Apache implementation of OSGI called *Felix*.

The rationale for the OSGI success is two fold. From a runtime perspective, OSGI is one of the most achieved platform for the dynamic management of components, called bundles in OSGI. Since 2003 and its adoption by the Eclipse community as the foundation of the Eclipse platform, OSGI enjoys good tool support. We considered both the runtime and tool perspective when we decided to

adopt the OSGI Technology for the design of JADE, we discuss both dimensions in the two sections below.

Runtime perspective

A component in OSGI is called a bundle; however, an OSGI bundle only partially corresponds to the Szyperski definition of a component in that a bundle only partially defines its dependencies. A bundle is in fact composed of two layers: a module layer and service layer. The module layer relates to modularity regarding Java types, leveraging Java class loaders. The service layer relates to the ability for bundles to register or look up services in a shared registry. The module layer is fully declarative through bundle manifests, as the Szyperski definition requires while the service layer follows a programmatic approach.

Corresponding to the Szyperski definition, a bundle is an independent entity that can be downloaded in an OSGI platform and dynamically assembled with other bundles already installed. This assembling is a two-phase process that corresponds to the two layers of a bundle—the module and the service layers. Once a bundle is installed, the OSGI platform automatically attempts to resolve the module of that bundle (there is a single module per bundle).

The module is meta-declared in the bundle manifest, written in a domain specific language that supports the definition of module dependencies. A sample bundle manifest is illustrated in Chapter 4, figure 4.1. A module is essentially grouping a set of Java packages, themselves grouping a set of Java types, both classes and interfaces. A module may export some of the Java packages that it has locally. A module may also import Java packages. It does so for essentially two reasons. One is to resolve local Java types, the imports therefore corresponds to missing Java types that are needed by local Java types to resolve at load time. The other reason is to provide a larger scope of Java types.

Module defines a visibility scope for Java types, like Java class loaders do. In fact, an OSGI module is a class loader at runtime in the Java Virtual Machine (JVM). It provides access to its local types, of course, but also to the types of its imported Java packages. This happens through the traditional class loading delegation between Java class loaders (Sheng and Bracha 1998). The Java type scope of a module is used by the service layer.

Per bundle, one may define a *bundle activator* that is started when its module is resolved. A module is resolved when it has all its imports satisfied. When started, the bundle activator is an automatically instantiated Java class from the module. It has therefore the type visibility of that module. The bundle activator acts as the *service-level constructor* of the bundle, it creates services that it registers and looks up the services that it needs.

Hence, OSGI has adopted a programmatic approach to services rather than a declarative one. In other words, service dependencies are not meta-declared and managed by a service-aware container following an Inversion of Control (IoC) pattern. OSGI only provides a shared registry that tracks registered services. A service is registered using a dictionary of key-value pairs. Leveraging this dictionary-based meta description of services, the registry allows bundles to lookup services through LDAP-like filters.

A bundle may register or revoke a service at any point in time. The correction of the programming model is ensured via lifecycle events about services being registered and unregistered. Anyone using a service must listen to these events and be prepared to relinquish that service whenever a revoke event

is received. This programming model works but represents a real burden on developers. Furthermore, the service layer lifecycle is slave to the module layer lifecycle. The OSGI platform automatically starts and stops bundle activators according to the module lifecycle. An activator is started only if its corresponding module is resolved, stopped otherwise. Modules also have a lifecyle, distinct from the service lifecycle. A module is resolved only if it has its dependencies resolved, that is, if all its imports are satisfied. Since bundles may be installed, uninstalled, and updated dynamically, that is, without shutting down the OSGI platform, modules may not be immediately resolved when installed and resolved modules may go unresolved at any time.

Tool perspective

The availability of tools is a decisive factor in the success of complex platforms such as OSGI. Today, developers can get help from either Maven or Eclipse. There are essentially two difficult issues with OSGI bundle development: managing imports and exports on the one hand and on the other hand publishing (releasing) developed bundles in order to make them available for deployment.

Both Eclipse and Maven provide automated support for managing imports and exports. A good thing since manually managing imports and exports rapidly proves tiresome and error-prone for most Java developers. Without automated support, a developer needs to track all its import statements in his or her Java classes, combine them, and maintain the list of all imported Java packages that are not locally available. These are the imports to be declared in the OSGI manifest. Automated support makes this an absolute no-brainer and totally safe.

Regarding publishing bundles, the OSGI Alliance promotes the use of OBR, the OSGI *Bundle Repository*. Bundles in OBR are identified by a string and have associated metadata in the form of key-value pairs. An extract from the OBR metadata is illustrated in Figure 7.1. Through this metadata, one can specify for example the bundle contents and dependencies between bundles. The OBR repository includes a resolver on such dependencies that allows to retrieve a transitive closure of bundles. In other words, if one wants to download a given bundle, the OBR permits to know what extra bundles are necessary to download for that one bundle to resolve. This is a valuable core service for the deployment of bundles.

7.1.2 About JULIA

Since JADE leverages the FRACTAL model, we need a Java incarnation of that model for complementing the modularity provided by the OSGI platform. Julia is a highly-optimized incarnation of the FRACTAL component model in Java, whose implementation is open-enough that it can be ported to the module layer of osgi.

JULIA provides a platform for dynamic assemblies of functional components, following a service-oriented paradigm. In JULIA, components are regular FRACTAL components developed in Java. Server interfaces are typed using Java interfaces, providing an adequate separation between the interface behaviors and the internal implementation of components. Client interfaces are essentially Java references to objects implementing the correct Java interfaces.

JULIA is an extremely well engineered incarnation of the FRACTAL component model. It heavily uses mixins and load-time weaving in order to limit the runtime overhead of components, especially

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<?xml-stylesheet type="text/xsl" href="obr2html.xsl"?>
<repository lastmodified="20071019162042.169" name="{\jade} Package Repository">
  <resource id="7" presentationname="Apache Wrapper"
    symbolicname="org.ow2.jasmine.jade.wrapper.apache"
    uri="./j2ee/org.ow2.jasmine.jade.wrapper.apache-2.0.0.jar"
    version="2.0.0">
    <description>{\jade} wrapper of Apache Http server</description>
    <size>11214</size>
    <category id="org.ow2.jasmine.jade.wrapper" />
    <capability name="bundle">
      <p n="manifestversion" v="2" />
      <p n="presentationname" v="Apache Wrapper" />
      <p n="symbolicname" v="org.ow2.jasmine.jade.wrapper.apache" />
      <p n="version" t="version" v="2.0.0" />
    </capability>
    <capability name="package">
      <p n="package" v="org.ow2.jasmine.jade.wrapper.apache" />
      <p n="uses:"
        v="org.ow2.jasmine.jade.fractal.util,
        org.ow2.jasmine.jade.fractal.api.control,org.objectweb.jasmine.jade.util,
        org.objectweb.fractal.api,org.objectweb.fractal.api.control"
      />
      <p n="version" t="version" v="2.0.0" />
    </capability>
    <require extend="false"
      filter="(&amp;(package=org.objectweb.asm)(version&gt;=0.0.0))"
      multiple="false" name="package" optional="false">
      Import package org.objectweb.asm
    </require>
    .. (the rest of requirements)
  </resource>
</repository>
```

Figure 7.1: An example of OBR metadata

optimizing the membrane. In JULIA, the different controllers are collapsed into a single object through mixins, avoiding most of the space and time overhead of having membranes. This is important because JULIA targets reflexive programming in that the running assembly of FRACTAL components may be introspected and reconfigured at runtime, through the meta operations of the controllers in the component membranes.

JULIA has focused solely on dynamic reconfiguration at the functional level of an application; it relies on the assumption that all the Java types needed by JULIA components are available in the classpath of the Java Virtual Machine (JVM). Cleverly though, JULIA offers the right hooks to introduce class loaders. These hooks were precious when we assembled the JULIA and OSGI middleware into a single platform that offered both dynamic modularity and dynamic assembly of FRACTAL components. We had the foundation for JADE, we needed to understand how to wrap legacy systems so that they can be deployed and managed through the component membranes.

7.2 Wrapping Legacy Systems

Wrapping legacy systems is a development task that is specific to the JADE environment. The goal is to create FRACTAL components that act as a gateway between the management world of JADE and the functional world of legacy systems. It is therefore easier to think about wrapping legacy systems as inducing a two-layer architecture, as depicted in Figure 7.2.

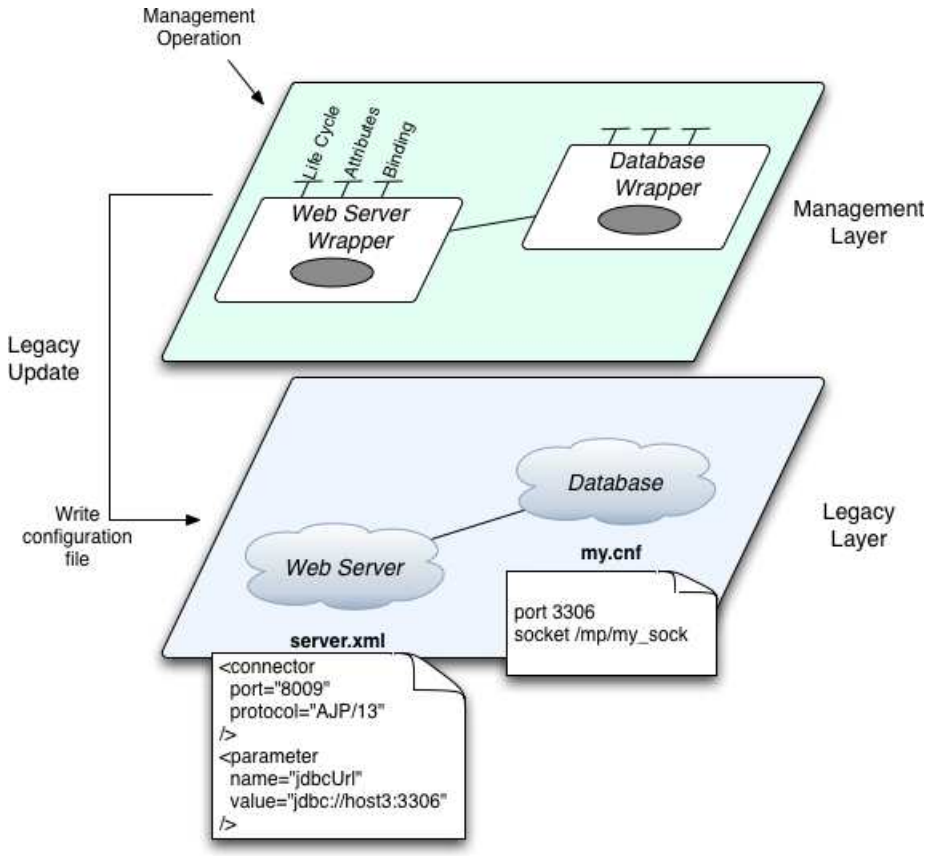


Figure 7.2: Basic Architecture of a Management System

In this example, we have two legacy elements to wrap: a web server and a database system. At the very least, the wrappers have to provide the following controllers. A lifecycle controller that enables to start and stop its legacy element. A binding controller that translates bindings into actual connections between legacy elements. In this example, this is done through parametrizing configuration files; the binding between the web server and the database is establishing through a JDBC connection on IPC port 3306.

Developers are usually creating one JADE component per legacy system, although nothing prevents to bundle legacy systems together. However, legacy systems are traditionally large and complex software systems that one would rather componentize than bundle together. Hence, a *wrapper component* usually reifies one managed legacy system as one JADE component, exposing the controllers defined by JADE. We discuss below the details of two real wrapping experiences: J2EE clustered Web

Application Server and an advanced JMS provider using a snowflake distributed design. For each, we give an evaluation of the challenges and difficulty of the wrapping process.

Web Application Server Scalable Web Application Server are often structured in multiple tiers as depicted in Figure 7.3 to meet the performance requirements of demanding web applications such as e-commerce and auctions, on-line banking, stock market quotation servers, web portals, and news servers.

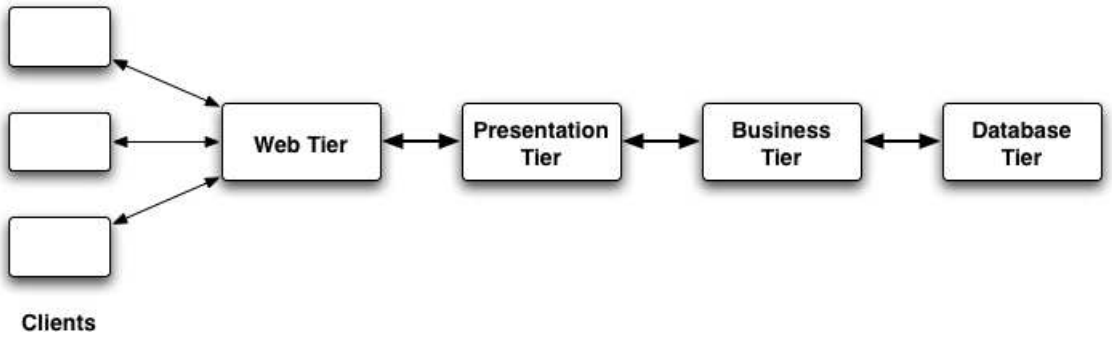


Figure 7.3: A clustered web application server

The *web tier* is an HTTP server, such as Apache. Its function is to receive and process the HTTP requests from clients. For static content, the *web tier* can answer the request directly. For dynamic content, requests are dispatched to the *presentation tier*, such as Tomcat Servlet engine. *Servlets* produce dynamic pages using the *business tier* (e.g. EJB Enterprise Java Beans) that implements the business logic, accessing the *database tier* (e.g. a MySQL server) that provides transactional query-oriented storage.

This n-tiers infrastructure is managed through a set of interconnected components that wrap the tiered legacy system. Each legacy wrapper provides the five controllers defined by JADE.

Attribute controllers are used to expose and change configuration attributes of the different tiers. For the web tier, it wraps the configuration file of the Apache HTTPD server. Hence, a modification of the port attribute of the Apache component is reflected in the *httpd.conf* file in which the port attribute is defined.

Binding controllers are used to reflect and manipulate connections between tiers. For instance, the Apache HTTPD server needs to be connected to the Tomcat Servlet engine. The implementation of this bind method is reflected at the legacy layer in the *worker.properties* file used to configure the connections between Apache and Tomcat servers. A bind operation will create that connection where an unbind operation removes that connection.

Lifecycle controllers are used to start and stop tiers. For instance, the web and presentation tiers can be started or stopped through the execution of shell scripts to start/stop the Apache HTTPD server or Tomcat Servlet engine.

Legacy wrappers are not components that can be transparently replicated by JADE, but they can nevertheless be replicated explicitly on a cluster of machines. Each instance of a legacy Apache HTTPD server and Tomcat Servlet engine would be wrapped as one primitive component, then paired

in a composite component representing one web server. The replicas of that web-server component on different machines would be grouped in a composite component capturing the replication group. Of course, this replication-oriented hierarchy of components combines itself with the deployment hierarchy when each machine of the cluster is reified as a node component that is a composite grouping all components deployed on that machine. A front-end switch that applies a round-robin policy across Apache servers would also be wrapped as a component with bindings that capture its connections to the different Apache HTTPD servers.

Message-Oriented Middleware Message-Oriented Middlewares (MOMs) are distributed platforms that enable a message-based integration of loosely-coupled heterogeneous distributed systems. We wrapped the MOM called JORAM (Java Open Reliable Asynchronous Messaging) that provides a fully compliant implementation of the JMS specification. JMS applications cooperate through messages, using either message queues for point-to-point communications or topics for a publish-subscribe paradigm. In this context, architecture-based management is two-fold. On the one hand, one needs to manage message queues and topics, something that is not covered by the JMS specification. On the other hand, one needs to manage the distributed snowflake architecture of the JORAM middleware itself.

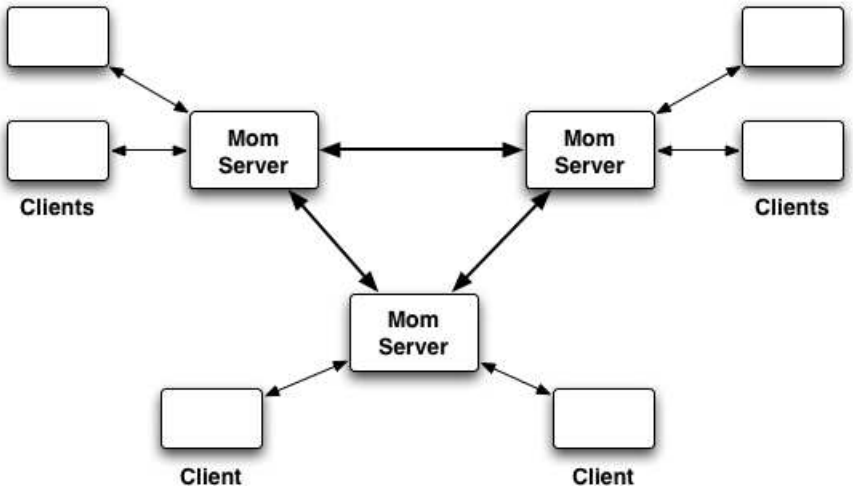


Figure 7.4: A snowflake infrastructure

The JORAM snowflake architecture is illustrated in Figure 7.4. It is a distributed middleware that sets up a routing overlay for delivering queue and topic messages efficiently and reliably. In JADE, JORAM servers are reified as composite components, each composite grouping all the middleware components deployed on the corresponding server. For instance, topics and queues are also wrapped as components and locally deployed on JORAM servers. The manifest file for the Joram JMS bundle is illustrated by figure 7.5.

Modelling message queues and topics as components allows for JMS administrative tasks to be done through JADE controllers. For example, one can create message queues or topics on certain JORAM servers. JORAM supports replicated topics that can easily be modelled and managed through

```
Bundle-Activator: fr.jade.osgi.joram.Activator
Bundle-Name: Joram Server
Import-Package: fr.jade.fractal.adl.attributes,
  ...
  org.objectweb.fractal.adl,
  ...
  org.objectweb.fractal.julia
Bundle-Description: A bundle that contains Joram components (Server, Queues,
Topics etc.)
Bundle-Vendor: Scalagent D.T.
Bundle-Version: 0.0.1
Manifest-Version: 1.0
Bundle-ClassPath: .,
  joram-mom.jar,
  joram-client.jar,
  joram-shared.jar,
  ow_monolog.jar,
  JCup.jar,
  jakarta-regexp-1.2.jar,
  jms.jar
```

Figure 7.5: An example bundle manifest file

JADE like we already discussed for replicated multi-tiered web servers. The same would be true of the JORAM advanced support for message queues with load-balancing capabilities. Through JADE, one can write queue managers that can create, deploy, and configure such message queues. Going further, JADE opens the path for more autonomic management functions such as autonomic load balancing for both message queues and replicated topics.

7.3 Component Deployment

Deploying components in JADE is a two-level process: *local deployment* and the *distributed deployment*. Local deployment is about the local installation of OSGI bundles, carrying both JADE wrappers and the wrapped legacy systems themselves or enough information to be able to trigger their installation through legacy installers such as DEB or RPM. Distributed deployment is about the distributed infrastructure that supports this local deployment on local OSGI platforms.

7.3.1 Distributed deployment

Before any deployment can take place within JADE, JADE itself needs to bootstrap, which is a somewhat autonomous distributed process. On a set of physical machines, a human administrator has installed the JADE local platform. Starting such a local platform creates a JADE *node*.

Each JADE node is a runtime instance of the OSGI platform that bootstraps individually. When an OSGI platform bootstraps, it starts a set of locally available bundles, the initial configuration. This initial configuration is described in an XML file automatically parsed by the OSGI platform. These bundles are usually self-sufficient so that they can be resolved autonomously, without requiring any further downloads of other bundles. This initial OSGI bootstrap produces a locally running JADE node that has minimal functionality. This minimal functionality is already an assembly of FRACTAL

components managed by JULIA, the OSGI platform providing the necessary modularity. This initial assembly, depicted in Figure 7.6, contains the following core JADE services:

- A OSGI management agent.
- A factory for FRACTAL components.
- A heartbeat generator.

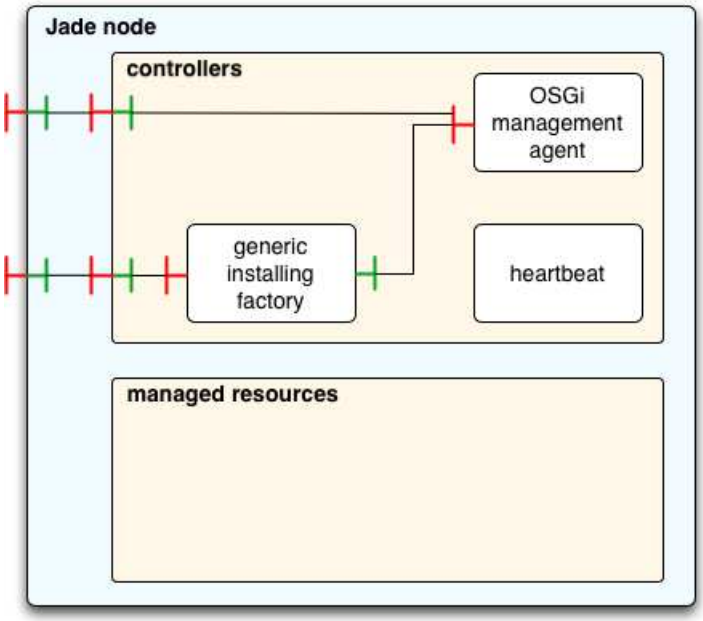


Figure 7.6: JADE Node

The management agent supports the local management of installed OSGI bundles, please refer to the section on local deployment below for further details.

The factory is an extended FRACTAL factory. As a regular factory, it supports the creation of FRACTAL components. In the JADE context, the factory is used remotely by the *autonomic brain* of JADE located on a specific administrator node, called the JADE *Boot*. The JADE Boot accesses local JADE node factories through FRACTALRMI, a FRACTAL-aware framework supporting Remote Method Invocations between distant FRACTAL components.

This remote use of local JADE node factories relies on an autonomous node discovery algorithm. On each bootstrapped JADE node, the heartbeat generator starts automatically and thereby generates heart beats. These heart beats, which are JMS messages, are automatically detected on the jade Boot by the *node discovery* component. When discovered, a JADE node is added to the *System Representation* of the managed distributed system.

The *System Representation* is an essential part of the JADE design that supports on JADE Boot the execution of autonomic managers such as the self-protection or the self-repair managers. The *System Representation* is essentially a causally consistent copy of managed components' membranes. In that, the *System Representation* offers a complete architectural view of the managed systems. It represents

locally on JADE boot the remote JADE nodes as FRACTAL composites. These local composites contain local copies of the membranes of all the wrappers deployed on remote JADE nodes.

Consequently, the *System Representation* provides a local representation of the system (hence its name) for autonomic managers to operate on. Managers may introspect the architecture and may also reconfigure it when necessary. Indeed, following the architecture-based management principle, any management task in JADE follows the pattern of introspection and then reconfiguration of the architecture. Autonomic managers never interact with the managed legacy systems, only with the copied membranes in the *System Representation*.

Of course, causality is enforced between this *System Representation* and the real wrappers in order to apply to the legacy systems the reconfigurations enacted by autonomic managers on the SR. This causality is achieved through remote method invocation that again rely on FRACTALRMI.

7.3.2 Local deployment

JADE uses Felix as an implementation of the OSGI specification from Apache. Felix implements the OSGI R4 specification, thus the most recent one at the time of writing of this document. However, nothing in JADE deployment is specific to Felix since we use only the official OSGI specification. As a validating proof of concept, we have also run JADE on Equinox, the OSGI implementation from the Eclipse community.

In this early design of JADE, we didn't extend the reflexive architecture of JADE with modules and physical packages, we had a local abstraction called a *local install package*. A local install package represented the physical entities underlying a locally installed JADE wrapper and the legacy system it wrapped. Using a Tomcat Servlet engine for example, a local install package would be a set of JAR files. One JAR file would be the OSGI bundle that contains the JADE wrapper written in Java as a JULIA component. It also contained the JAR files necessary to launch a Tomcat Servlet engine in a Java Runtime Environment (JRE).

Local deployment is therefore two-fold. One fold is about the wrapper side of the world and the other fold is about the legacy systems that are wrapped. It is the responsibility of the local JADE node to coordinate both sides, following the reconfiguration requests emanating from autonomic managers. This coordination takes place within the local factory on JADE nodes. When a local factory is requested to create a local wrapper, the factory has to take the following steps.

The factory has to identify the corresponding physical package. This information is given as part of the request to create a wrapper under the form of a physical package identifier. If the physical package is not locally present already, it is downloaded from a repository through HTTP. Once the physical package is installed, the factory has to extract the OSGI bundle from it. With the JAR file of the OSGI bundle, it can call the OSGI APIs to install a bundle. It is the OSGI platform that will open up the JAR file and, from the extracted OSGI manifest, will actually create the bundle in memory. Once created, the module of the bundle is automatically resolved by the OSGI platform (if it has all its dependencies met) and it is finally started. As a side effect of being started, the actual creation of the wrapper as a FRACTAL component managed by JULIA happens, using the Java classes provided by the module of the bundle.

The JADE node factory is an extended FRACTAL factory because it keeps track of the components it created, as described in Section 6.3.4. This allows a JADE node to keep track of the wrappers that

have been locally deployed. However, this creates a local root of persistence for locally deployed wrappers, both in memory and on disk. We therefore extended the factory API to include a undeploy operation that allows autonomic managers to request that a JADE node undeploys and forgets about a locally known wrapper.

This undeploy operation concerns of course not only the wrapper but also the corresponding legacy system that is wrapped. Both are removed. The OSGI bundle is uninstalled from the running OSGI platform, thereby cleaning the in-memory footprint of the deployed wrapper. The physical package is also removed, thereby freeing on-disk storage. An important point needs to be discussed though: physical packages may be shared in the sense that they may contain several wrappers. The rationale is that it is advantageous for wrappers wrapping small and cooperating legacy systems to be bundled together as one downloadable unit.

However, this requires a more subtle handling of the undeploy semantics. The undeploy request comes on a single wrapper, which is all what autonomic managers know about. Hence, the local JADE factory has to put in place a reference counting scheme to know when a physical package has no instantiated wrappers in the OSGI platform. Only then could the actual OSGI bundle be uninstalled and the physical package removed from disk.

As described previously, the module abstraction with which local JADE factory works is called the *LocalInstallPackage*. A Java signature of this interface is illustrated in figure 7.7

```
public interface LocalInstallPackage {

    public Loader getLoader();

    public int getReferenceCounter();

    public void increaseReferenceCounter();

    public void decreaseReferenceCounter();

    public List getDependencies();

    public PackageDescription getPackageDescription();

    public boolean isMarked();

    public void unInstall();
}
```

Figure 7.7: The Java signature of the *LocalInstallPackage* interface

This abstraction is the central piece of the local deployment infrastructure in the first version of JADE. It represents a module – an OSGI bundle – within the explicit architecture managed by JADE. It provides the JULIA framework with a *Loader* encapsulated by a given bundle. This loader is used to instantiate functional components. Every local install package contains information on the number of other local install packages which depend on the given one. This information can be obtained by calling the *getReferenceCounter* and is used by the garbage collector to remove unnecessary modules. However, modules are only removed if they are not “marked”. Information whether a module is marked or not is provided via the *isMarked* method. The possibility to mark local install packages is

useful especially if such a package is large and its installation on the local file system is costly. Marking such packages prevents them from being garbage collected and thus optimizes the deployment time. A *PackageDescription* is the description used to identify a package within a possibly remote and distributed repository. This description is used at the deployment time, but can also be obtained from a local install package after the deployment completes, by calling the *getPackageDescription* method of the local install package.

7.4 Admin Console

Very early on, it was obvious that JADE needed to provide an administration console. Right from the start in fact since an initial deployment had to take place, creating an initial architecture that autonomic managers could then observe and manipulate.

The deployment engine is a service of the JADE Boot that supports the admin console. The administrator, through the console, can upload an ADL description of an architecture to create or can directly use script commands. Currently BeanShell and FScript scripts are supported by JADE. Using scripts and an interactive console, a human administrator can introspect and reconfigure the architecture. In contrast, the upload of an ADL-based deployment is more suited for the initial deployment of the system. In the current implementation, the JADE deployment engine is a largely modified Fractal ADL factory[1], as represented by figure 7.8

Therefore, the deployment engine is a component-based application and is part of the Jade's explicit architecture. It consists of three composite components: (1) loader, (2) compiler and (3) backend each of them containing a set of primitive components. The loader composite component is responsible for verifying the correctness of the JADE deployment file. Compiler component creates the deployment plan tasks that are executed by the backend component. The compiler component consists of several primitive components, which are executed in the top-down order, therefore the module compiler for example is executed before the type compiler.

The parsing of the uploaded ADL description is performed within the JADE Boot. The JADE Architecture Description Language (ADL) is an extension of the FRACTAL ADL. Therefore, it is XML-based and provides a static description of the system to be deployed. It contains all the information needed by the deployment system to deploy a given JADE-enabled application. A minimum set of such information is the following:

- Architecture of the application to be deployed i.e. components and their relation in terms of hierarchy and interconnections
- Configuration of the components — values of their attributes
- Placement information, i.e. on which machine which component is to be deployed, or certain constraints on component co-location, without explicit information on the target nodes
- Information on modules used by the components, i.e. which module contains the code and other resources needed by the given component

[1] *http://fractal.objectweb.org/current/doc/javadoc/fractal-adl/overview-summary.html*

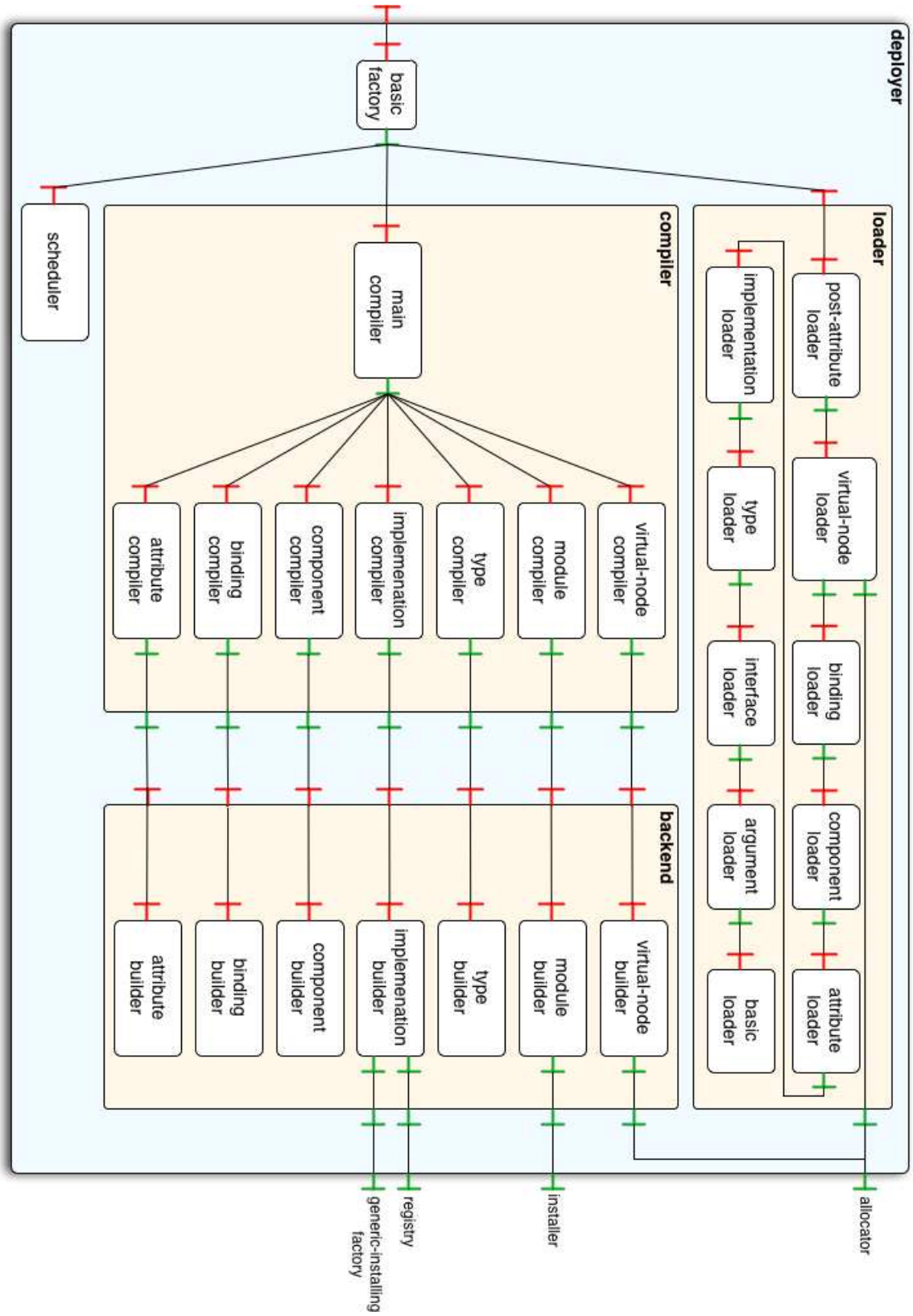


Figure 7.8: JADE deployment engine

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
  "classpath://fr/jade/service/deployer/adl/xml/jade.dtd">

<definition name="J2EE">
  <interface name="service" role="server"
    signature="fr.jade.service.Service" />

  <component name="start"
    definition="fr.jade.resource.start.StartType">
    <virtual-node name="node1" />
  </component>

  <component name="apache"
    definition="fr.jade.resource.j2ee.apache.ApacheResourceType">
    <attributes
      signature="fr.jade.fractal.api.control.GenericAttributeController">
      <attribute name="user" value="jkornas" />
      <attribute name="port" value="8081" />
    </attributes>
    <virtual-node name="node1" />
    <module name="Apache Wrapper" />
  </component>

  <component name="tomcat"
    definition="fr.jade.resource.j2ee.tomcat.TomcatResourceType">
    <attributes
      signature="fr.jade.fractal.api.control.GenericAttributeController">
      <attribute name="workerPort" value="8098" />
    </attributes>
    <virtual-node name="node2" />
    <module name="Tomcat Wrapper" />
  </component>

  <component name="mysql"
    definition="fr.jade.resource.j2ee.mysql.MysqlResourceType">
    <attributes
      signature="fr.jade.fractal.api.control.GenericAttributeController">
      <attribute name="user" value="jkornas" />
    </attributes>
    <virtual-node name="node3" />
    <module name="MySql (linux x86)" />
  </component>

  <binding client="apache.worker" server="tomcat.resource" />
  <binding client="tomcat.jdbc"   server="mysql.resource" />

  <binding client="start.rsrc_mysql"   server="mysql.resource" />
  <binding client="start.rsrc_tomcat" server="tomcat.resource" />
  <binding client="start.rsrc_apache"  server="apache.resource" />

  <virtual-node name="node1" />
</definition>
```

Figure 7.9: An example JADE deployment file

An example JADE deployment file is presented in figure 7.9, it builds a simple 3-tier Web Application Server, composed of Apache, Tomcat and MySQL legacy servers. As specified by the virtual-

node tag, each of the tiers is deployed on a separate target machine. The virtual-node tag provides only collocation information, i.e. it does not provide information on the exact name or IP address of the target machine, but only says which components should be placed together, and which should not.

The only ”dynamic” aspect of this description is the order in which the tiers are started. MySQL needs to be started before Tomcat, which in turn needs to be launched before the Apache server. Since Fractal by default does not allow to specify the order in which components are started, JADE uses a specific component, called *start*, to achieve this goal. The *start* component launches all the components bound to it in an order equal to the one of bindings. Therefore, in the example above, starter will first launch MySQL, then Tomcat and finally Apache.

Information about component modules is provided through the *module* XML element. Each component specifies one module element. Depending on the implementation of the module repository from which the modules are obtained, module identifiers can have different forms. At present we reuse the identifiers from OSGI Bundle Repository (OBR), as will be explained later in this section.

7.5 First Evaluation

JASMINE is an open-source project aiming at providing an administration tool for various types of middleware, such as Java EE servers, Message-Oriented Middleware platforms, SOA containers etc. The goal of the project is to facilitate the life of administrator of these heterogeneous systems. JASMINE relies on the deployment infrastructure provided by the OSGI-based version of JADE. On top of this infrastructure, it provides a graphical user interface (GUI) built using the Eclipse RTP technology. This graphical interface, illustrated in figure 7.10 supports the drag-and-drop style assembly of middleware components and the deployment of these software components on potentially distributed target machines, which are also represented graphically as components within the JASMINE console. Furthermore, using the JADE’s deployment infrastructure the JASMINE builds an autonomic management system on top of it, based on the Drools business rule management system (BSRM).

Once a middleware configuration is defined by an administrator using the JASMINE console, it can either be saved for further processing later on, or it can be given to the JADE’s deployment engine for deployment. If the configuration is given to JADE, it is first parsed into the XML form, like the one illustrated in figure 7.9. Next, it is converted into a deployment plan and deployed accordingly to the JADE’s deployment process.

Jasmine use-case: Component-based JOnAS [2]

JonasALaCarte (Abdellatif 2005) is an example of deployment of a component-based middleware platform performed with JADE. It is a component-based version of the JOnAS Java EE server, conforming to the JADE’s component model and packaged into OSGI bundles. The work on deploying JonasALaCarte with JADE has been described in (Abdellatif et al. 2007).

To create the component-based version of JOnAS we have replaced all the static interfaces used for communication between JOnAS entities (services and management entities) with inter-component bindings. As illustrated in figure 7.11 the JonasALaCarte server is therefore a composite component

[2]The work on deployment of a component-based JOnAS (JonasALaCarte) has been performed in collaboration with Takoua Abdellatif.

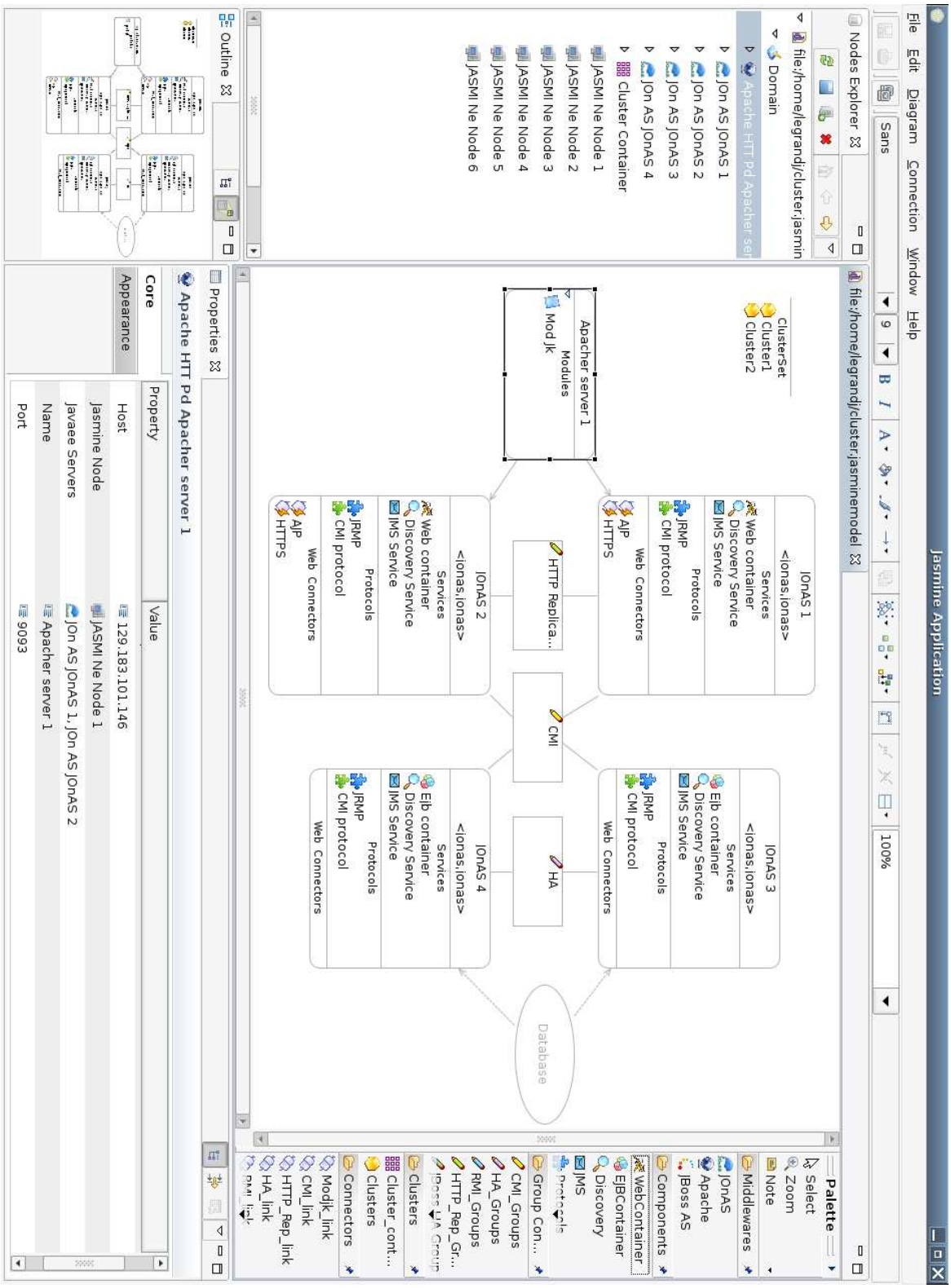


Figure 7.10: Jasmine assembly and deployment console

encapsulating a set of primitive components which represent the JOnAS services bound to one another. These bindings can potentially be remote, in which case we obtain a distributed application server. Each service is a separate component that exposes a basic set of control interfaces defined by JADE, namely:

- The attribute controller. In case of JOnAS services, this controller sets the service configuration values
- The life-cycle controller, which is implemented following the JSR77 (*J2EE Management Specification (JSR77)* 2005) life-cycle graph. This controller allows for the modification of the state of JOnAS services
- The binding controller, which establishes references between JOnAS services

The binding to the configuration manager component lets each service obtain a default configuration. Additionally, each component exposes a set of specific functional interfaces. For example, the EAR service implements deployment interfaces compatible with JSR88 (see Section 2.3).

The ADL describes the configuration of each service and lists its interfaces.

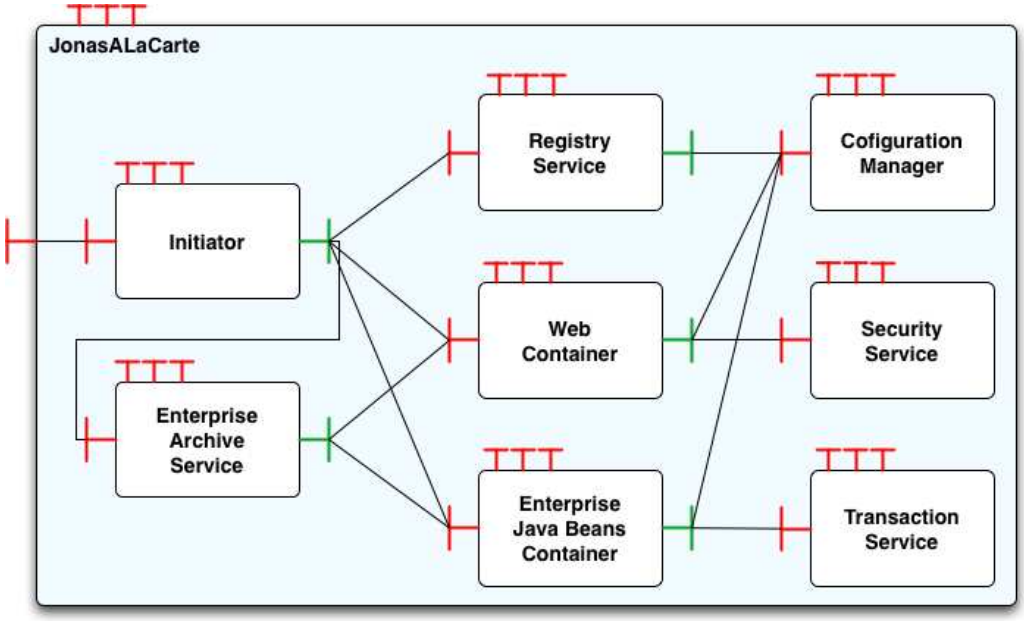


Figure 7.11: The architecture of the JonasALaCarte J2EE server.

After the "componentization" step, we packaged each JOnAS component in an OSGI bundle. Every such bundle contains the component's code. In addition to component bundles, an interface bundle contains all the interfaces involved in the communication between the different application server components. Unlike "regular" JOnAS, which delivers all services in a single package, JonasALaCarte packages each service separately. This decreases the memory footprint of JOnAS on the target machines and reduces the deployment time of the whole server within, for example, a cluster.

To deploy clustered JonasALaCarte, an administrator must either create the deployment configuration within JASMINE or write the ADL file manually. For clustered environments, the configuration must specify not only the architecture of the server (that is what services are to be deployed in a given configuration), but also the information on collocation of the components building the server (that is, the virtual nodes on which the different services are to be deployed). The JADE's deployment engine then automatically deploys the entire clustered architecture. Unlike in "regular" JOnAS clusters, the unit of replication in JonasALaCarte is the service component, not the whole server. This selective replication is important because the Enterprise JavaBeans (EJB) containers and Web containers are generally execution bottlenecks, so we need more replicas for these services than for other ones, such as registry or transaction services).

As with any component-based software deployed and managed by JADE, in JonasALaCarte we abstract an application-server's cluster deployment and configuration to the uniform handling of components. A cluster configuration is therefore a particular configuration of the application server, where components are distributed and replicated on different Java Virtual Machines. We use the same management tools to manage a standalone server in a single JVM as we do to manage a cluster of servers.

Compared to traditional scripts, writing the deployment plan with ADL or creating it within JASMINE is less error prone, because the overall system architecture is exposed to the administrator at different levels of granularity. The syntax of the configuration language or the configuration tools used for JOnAS, Tomcat, Apache, and so forth becomes homogeneous. Detecting configuration errors is easier and the deployment time is significantly shorter.

Using the local installation mechanism, the server administrator can also perform some basic dynamic updates of components. This feature may be of use when updating JOnAS services due to the release of new versions or for performance optimization. However, in the context of server-side Java EE middleware, we need to consider two additional aspects of dynamic updates of software: the treatment of the ongoing requests from clients to the server during the reconfiguration and the state of the modified service and its impact on the other services. Regarding the ongoing requests, we implemented a front-end proxy that queues the incoming client requests during the reconfiguration time (evaluated in advance). The state depends on the service component being dynamically added to or removed from the server and on the applied management policy. These aspects of reconfigurations as well as proposed solutions to address them have been described in (Abdellatif 2005).

From the update sequence point of view, the (autonomic) manager of JonasALaCarte first stops the component that needs to be updated. This translates to the stop operation performed on the corresponding managed elements life-cycle controller. Next, the manager unbinds this component from other components, which translates to the unbind operation on this components binding controller and calls the update interface with the new package. Finally, the manager updates the component and re-establishes the bindings, and the updated version of the component treats the requests in the queue.

As a runtime configuration, consider the case of a fault-recovery policy. This policy is similar to the micro reboot policy pioneered by JAGR (JBoss with Application-Generic Recovery) (Candea et al. 2003) in the context of J2EE servers (although at the level of J2EE applications, not of the J2EE server itself). The policy consists of rebooting only the necessary services in case of failures and not the whole server.

Now consider the failure of a node containing a Web container – an interesting issue in the context

of internet services (Oppenheimer et al. 2003). For fault tolerance, an in-memory session replication for HTTP sessions is performed on a separate machine for each Web component modification or update. This technique is delivered with the Web container (Tomcat) and is classically used in J2EE clusters. Upon being notified of the node failure, the JADE's fault manager retrieves from the system representation the software configuration that was running on the failed node. It then requests the allocation of a new node, considering some hardware criteria (for example CPU charge and memory usage). The fault manager knows of available nodes because every new node registers itself within the management system, under a well known address. Once the node is allocated from the list of avaiable ones, the fault manager deploys a Web container component on a newly allocated node and to reintegrates it into the software architecture in terms of composition and binding. Finally, the repaired web container can be started. If an application state must be recovered, when starting the application components in the new container, the fault manager component sends a request to the replica group to get the session state.

7.6 Conclusion

In this chapter, we presented the first design of the JADE platform. JADE adopted an architecture-based approach to system management, modelling the managed legacy systems as FRACTAL components. My work was the design and implementation of the deployment capabilities of JADE.

Deployment happens both at a distributed and local level within JADE. Indeed, the managed legacy systems need to be deployed on various physical nodes of the underlying distributed system. Furthermore, the control of these legacy systems also has to be deployed, that is, the JADE wrappers developed as FRACTAL components developed with JULIA that provides a Java incarnation of the FRACTAL component model. Hence, deployment in JADE is a process that happens across the Java and legacy worlds.

For the legacy world, any existing legacy deployment infrastructure such as RPM or DEB or Windows Installer could be used. In fact, most legacy systems are deployed today using these infrastructures. The only requirement from our side is that these infrastructures must provide a programmatic API so that they can be driven from within the JADE platform, which is something that most do. Also, in many cases, legacy systems could be just wrapped within one JAR file along with the code of the wrapper, both downloaded as one unique OSGI bundle; but this is only a simplification, not a requirement.

For the Java world, we decided to use the OSGI platform. It provides advanced modularity for the managed of dynamically assembled components. These components are called bundles and can be downloaded through the network. OSGI seemed a great foundation to build on, especially that we were able to adapt JULIA to run on top of OSGI modularity. The result was a complete foundation for the local deployment of JADE wrappers. That foundation included a Java Runtime Environment for the portability of the wrapper code and the ubiquity of the JADE node. It also included the OSGI and JULIA frameworks to provide modular and dynamic assemblies of components that supported well the deployment needs of autonomic managers.

This design of JADE has been used successfully in different environments, such as the management of clustered Web Application Servers, databases, and Message-Oriented Middleware. It is the

foundation of the Jasmine open-source project that aims at providing advanced management capabilities to the ObjectWeb community. However, this first design of JADE is fundamentally based on the assumption that components are used as wrappers of legacy systems. This assumption is perfectly valid in the previous use cases but limits the ability for JADE to manage itself, something that seemed more and more appealing as we realized that JADE itself was becoming a component-oriented distributed system and would soon require very similar management facilities as the ones it provides for legacy systems.

This realization slowly matured into the new design of JADE presented in the next chapter that corresponds to the novel approach of JADE detailed in Chapter 8. This novel approach considers JADE as offering a distributed component-oriented platform that provides a full reflexive architecture that can be introspected and reconfigured through a fault-tolerant distributed programming model. This required changing fundamentally our design and opened up challenging research questions. This research work is still on-going at the time of this writing.

Résumé de chaptire 8

Dans le chapitre 8 de cette thèse on décrit une nouvelle conception de JADE. Cette version existe car dans le chapitre 7 on a identifie des tensions fondamentales entre JULIA et OSGI au sein du JADE. On décrit ces tensions dans la section 8.1 de ce chapitre. Ensuite, dans la section 8.2 on explique la nouvelle architecture reparti de JADE. Puis dans la section 8.3 on descends vers le système a modules qu'on a conu et prototypé pour cette nouvelle version de JADE. On se focalise sur les spécificités de résolution des modules dans la section 8.3.3 et sur la optimisation des mises a jour des modules, une opération important dans les systèmes réflexives dynamiques, dans la section 8.3.4. Dans la section 8.4 on présente les paquetages physiques, qui peuvent łtre téléchargé et installé et qui contiennent les implantations physiques des composants. On se focalise sur le ramasse miettes dans la section 8.5, avant de conclure ce chapitre dans la section 8.6.

Chapter 8

JADE: Second Design

Contents

The *new* JADE has stepped in the world of a reflexive and recursive design. We kept our reflexive design to architecture-based system management that leverages the reflexive component model of FRACTAL. Beyond this first design choice, we revisited our first design and pushed further a recursive design where JADE is designed and developed using JADE technology.

JADE technology is a framework to build autonomic systems, to be contrasted with building a management system that aims to provide autonomic properties to managed legacy systems. This is not to say that the new approach could not be used to manage legacy systems. In fact, the same wrapping technology could be used. But the essential difference is that the use of the FRACTAL component model is no longer limited to building and deploying such wrappers.

The new goal is much more ambitious. Designing and developing it became clear that writing autonomic managers required that almost all autonomic managers applied to themselves. In other words, JADE needed to manage itself. For instance, JADE needed to deploy itself, not only what it managed. The self-repair manager also needed to self repair to provide real autonomic behavior. A self-protect manager that protects managed components could also be the subject of attacks and needs to self protect.

This requires evolving the design of JADE towards a foundation to build such autonomic managers. The foundation is a distributed platform that still advocates a reflexive component-oriented paradigm where introspecting and reconfiguring the reflexive architecture is fault-tolerant, that is, both atomic

and highly available through replication. We approached this design with a recursive design philosophy, searching to provide enough self-behavior through the very own technology we were trying to build.

We decided to keep our starting point in Java for it provides a portability and ubiquity that is hard to beat. However, we decided to do without the OSGI platform whose architecture and design conflicts with ours. Similarly, we needed more freedom in the design of the incarnation of the FRACTAL component model than JULIA permitted. The rationales emerged from the two first self-behaviors that needed to be provided: self-deployment and self-repair.

In this chapter, we present the new design of JADE, focusing on the JADE foundation. In Section 8.1, we discuss the breaking point of the previous design of JADE. Section 8.2 gives an overview of the new, distributed architecture of JADE. In Section 8.3, we present our design for modularity in Java. We focus on the specifics of module resolution in Section 8.3.3 and on optimizing module updates, an important operation in dynamic reflexive systems, in Section 8.3.4. In Section 8.4, we present our physical packages that support physical download and caching of the implementations of components. We focus on the garbage collection issue of physical packages in Section 8.5, before we conclude this Chapter in Section 8.6.

8.1 Evaluation—Breaking Point

In this section we come back on the first design of JADE and we propose an analysis of where it breaks down with respect of our new approach for JADE that is both reflexive and recursive. One of the main issues is a fundamental architectural mismatch between the decision flow of autonomic management in JADE and the OSGI design. By essence, an architecture-based approach is a top-down decision flow. Autonomic managers observe a running system and react to changing conditions by reconfiguring the architecture of that running system. This means that the policies are above, in the managers, and the driven mechanisms are below, in the runtime platform.

In the case of deployment, the conflict shows in both the distributed and local dimensions of deployment. The conflict with respect to distributed deployment is not really directly with the use of the OSGI platform, it is with the use of the OBR repository. The conflict with respect to the local deployment results from incompatible architecture perspectives between the OSGI specification and an architecture-based approach à la JADE.

8.1.1 Distributed deployment conflicts

The use of OBR is dual. On the one hand, the OBR is not really a repository for publishing bundles. Consequently, the OBR does not support an ecosystem around OSGI bundles—there is no market place for OSGI bundles. The OBR is simply a way to publish a description of modules and their dependencies. This description is simply an XML description that a user of OBR would have to download and locally process using the OBR resolver. If this works for a few bundles up to a few thousands (if one has a powerful machine), it certainly does not scale to a world-wide ecosystem of OSGI bundles. Hence, OBR gives very little above a dependency resolver that we would need to be part of an autonomic self-deployment manager anyway.

There is one idea to retain though from OBR, the ability to have suggestions about what modules would be necessary to resolve a given module. Given a module, OBR is able to process bundle dependencies (from the locally downloaded description) and provide a transitive-closure download list. By transitive closure, we mean that given a bundle that one needs, OBR can propose a list of bundles to download that downloaded altogether satisfy all the dependencies of that bundle. This is an important of a solution because module imports are flexible dependencies in that they specify what the needed Java packages are and not where they should come from. Hence, we need a brokering infrastructure that can answer which modules could potentially fulfil an unresolved import.

In of itself, using the OBR seemed a good idea, as it provided a useful functionality. In the context of JADE, however, this has proved to be a mistake because it divides the autonomic decision between JADE and OBR, without any consistency guarantee. On the one hand, OBR decides to pick certain OSGI bundles looking at module-level dependencies. On the other hand, JADE looks at service-level dependencies between FRACTAL components. Both dependencies are related because service-level dependencies do mention language types and the type correctness of the assembly depends on the consistency between module-level and service-level dependencies.

A solution would be to correlate both decision making, by maybe adding service dependencies to the dependencies that OBR processes, but this is awkward at best. The problem is that OBR has only a partial view of the dependencies anyway. It does not capture the architecture of the overall distributed system and does not know about other constraints on modules such as co-location hints from the deployment plan. Therefore, in choosing different versions of modules, it is blind to some dependencies and incompatibilities of the desired assembly. Basically this means that JADE cannot rely on OBR. Besides requiring a development effort within JADE, there is no show stoppers here. The point is simply that the entire decision process of choosing and downloading modules and their related physical packages must be done within JADE, using the full knowledge of the reflexive architecture and of the deployment plan.

8.1.2 Local deployment conflicts

This leads us to the show-stopper conflict at the local deployment level between OSGI and JADE. As we just discussed, JADE has to be in control of the overall deployment plan, which includes not only which physical packages to download and where, but it also includes the binding decisions between modules. The crucial point is that the creation of the bindings carries the overall consistency of the distributed assembly of components, including both regular components and modules. And this is exactly where the use of OSGI breaks down.

The problem is that there is no way for JADE to impose those bindings to OSGI, the OSGI platform autonomously decides how modules should be resolved, resolving module imports and exports as it sees fit. Of course, the resulting bindings will be correct from a module perspective, respecting the semantics of OSGI and therefore the type system of Java, but there is no longer any guarantee that the bindings between regular FRACTAL components are consistent. This is especially true in the case of the Release 4 of OSGI that allows for multiple versions to be retained by the OSGI resolver, potentially partitioning Java types.

To put it simply, we found that building an autonomic layer on top of another autonomic layer just does not work if there are subtle dependencies in between the decision processes. At first, JULIA and

OSGI seems just perfectly complementary with the OSGI platform providing modularity and JULIA providing a service-oriented paradigm for FRACTAL components. We all missed the consistency requirements between service-level bindings and the type-level bindings.

As a concrete example such a consistency problem can be illustrated by figure 8.1. In the presented scenario, functional components *A* and *B* have a runtime binding, resolved by JADE. Via this binding they exchange a number of types, including the type *T*. However, due to the lack of a proper import/export declaration between the OSGI bundles (modules) providing types for components *A* and *B*, the OSGI resolver does not establish a module-level binding between the two bundles. As a result, component *A* sees a different type than component *B* and the whole configuration is inconsistent. At runtime, when the two components attempt to use the shared type *T*, the Java Virtual Machine will raise a *ClassCast* Java exception.

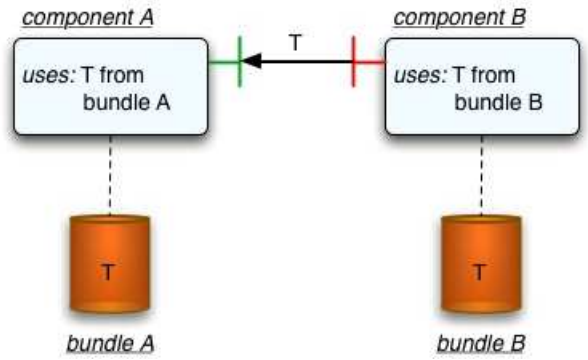


Figure 8.1: An example of incoherent configuration

Beyond this "static" configuration scenario, the conflicts in the local deployment layer of JADE extend to the lifecycle of components. When JADE completes a reconfiguration plan, it does so including the lifecycle transitions that are necessary to permit the reconfiguration. For instance, JADE computes the transitive closure of components that need to be stopped, including those that need to be disposed of. For instance, if a module needs to be updated, JADE computes the transitive closure of all modules depending on that module and will include tasks in the plan to stop all of them. This requires a precise and accurate knowledge of the bindings amongst modules, which is impossible when using an OSGI platform.

In fact, the OSGI platform does the same decision making autonomously, computing its own transitive closure, using an implementation-specific algorithm that cannot be predicted as it is not specified by the OSGI Alliance. This translates into spurious lifecycle events at the level of OSGI bundles requesting them to stop. This is obviously correct for applications fully adopting the OSGI programming model, but JULIA is not. In fact, JULIA is not aware of OSGI modules on the one hand and on the other hand does not use OSGI services. JULIA is therefore ignorant of OSGI lifecycle events. The result will be an inconsistent system, leading undoubtedly to runtime failures.

A concrete example of this type of a conflict is illustrated in figure 8.2. In this scenario, component *A* is updated by the JADE framework. Since JADE manages dependencies between functional components and their corresponding OSGI modules (bundles), the JADE's autonomic manager asks the OSGI platform to update the *bundle A* which provides types for functional component *A*. However,

during an update of *bundle A* OSGI takes an internal decision to also update *bundle B*. This is because *bundle B* depends on *bundle A* by importing Java types from it. OSGI's resolver's algorithm imposes an update of the dependent bundle in such a case, due to the way types are resolved by the JVM. This raises a conflict between the update manager of JADE and the resolver of OSGI—any functional component created from *bundle B* should be recreated when this bundle is updated. In our case this means that component *B* should be recreated. However, JADE is unaware of the fact that *bundle B* was updated and thus attempts to rebind the "old" version of component *B* on the "new" version of component *A*. This results in potential type inconsistencies in terms of what these two components exchange. Furthermore, since type resolution in Java is lazy and OSGI decided to discard the "old" version of *bundle B*, which in our case component *B* continues to use, further execution of component *B* after an update may result in it being unable to resolve Java types that it needs to operate correctly.

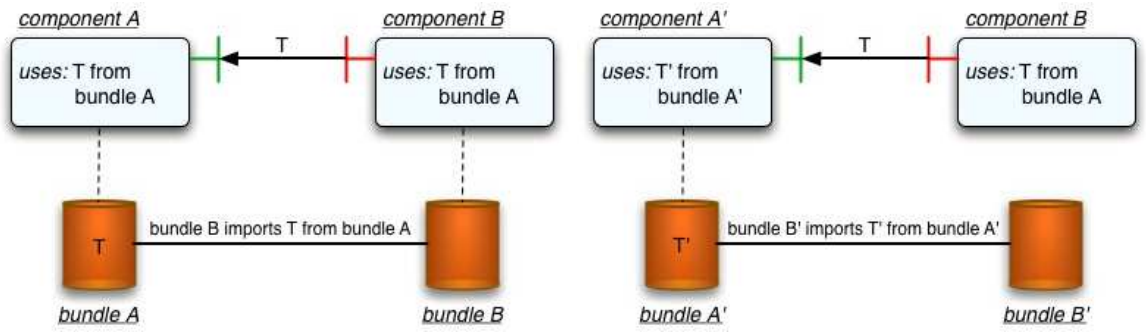


Figure 8.2: Type incoherence after a component update

One may feel that a solution would be to adopt a design similar to what the Spring-OSGI framework has chosen (Framework 2006). It would indeed solve some of the problems but not all for the distributed context of JADE. The Spring Framework (Community 2004) is not unlike JULIA in many regards, it provides a Java framework for dynamic assemblies of components. It is extremely successful in the J2EE server arena, abstracting with components the various complex frameworks of Web Application Servers.

Recently, the Spring Framework has adopted the OSGI platform as its foundation, like Eclipse did a few years earlier. The rationales are the same for Spring as they were for Eclipse or for JADE, the OSGI platform is one of the most advanced platforms when it comes to combining modularity and services. The early approach of Spring was consistent with the design we adopted in JADE: keep the service layer between Spring components as it is and leverage the underlying modularity provided by the OSGI platform.

The conclusions were essentially the same as ours—this design did not work. A second design followed, which is the current Spring-OSGI design where both layers were integrated. In fact, the module layer is the master and the service layer the slave. Any Spring service exported by a Spring component is in fact an OSGI service, registered in the dynamic OSGI service registry. The Spring runtime fully understands and follows the lifecycle events from the underlying OSGI platform, obeying to events requesting to drop the use of services as well as events notifying that components as a whole must be disposed of because a module is going unresolved.

A similar design would work for JULIA, but conflict with the FRACTAL model. Indeed, controllers have the complete control of the lifecycle and bindings in FRACTAL. A JADE specific implementation could have a restricted approach, using hard-coded controllers that in fact obey the service and module lifecycle events from the OSGI platform. But this would only work in a centralized system, with a unique OSGI platform making decisions.

For JADE, the Spring-OSGI design is not an option because of its distributed nature. Components are distributed across nodes and therefore across different runtime instances of the OSGI platform. Therefore, JADE must globally coordinate reconfigurations of the architecture. For instance, it is impossible to have the two ends of a remote bindings making different decisions about the version of the Java types to use for the type of the service interface. Moreover, remote components must be stopped for the same reasons as local ones must. As far as our understanding goes, using OSGI is not an option for a reflexive architecture-based management system like JADE.

Regarding JULIA, the decision to stop using it was more difficult to make. Julia is an efficient incarnation of FRACTAL in Java, always something one wishes to have. However, the same optimization that ensures JULIA performance is also the source of real development complexity. Just debugging alone across the use of JULIA and FRACTALRMI is awkward at best, seriously hindering productivity and our ability to seek for the best trade-off in the context of JADE. This lack of freedom in the implementation details of the FRACTAL model was the final decision maker. We already had to extend JULIA several times for accounting for bi-directional bindings, component tracking factories, or introducing class loaders on top of the internal JULIA ones. Moreover, we also made changes to FRACTALRMI in order to make it module-aware and we expect further changes around the centralized registry, not an option for a distributed autonomic system.

8.2 Distributed Architecture

The new design has evolved towards a truly distributed architecture without any single point of failure. In particular, the JADE Boot has been removed as a singular entity and replaced by replicated components. In other words, the autonomic self-deployment manager is now a replicated component, like the autonomic self-repair manager.

The bootstap sequence is now entirely autonomic, after the initial human intervention to install the JADE TFTP-like bootstrap loader on certain physical machines. This very primitive loader knows how to contact known servers to download the first image of the JADE platform. We are still using Java, so the platform could reuse an already installed JRE or download one. Once Java is available, a certain number of physical packages are downloaded and installed locally in the local repository. These very first physical packages correspond to the initial component assembly that makes up a JADE node.

This initial assembly, depicted in Figure 8.3, contains the following core JADE services:

- A JMS client.
- A factory for FRACTAL components.
- A heartbeat generator.

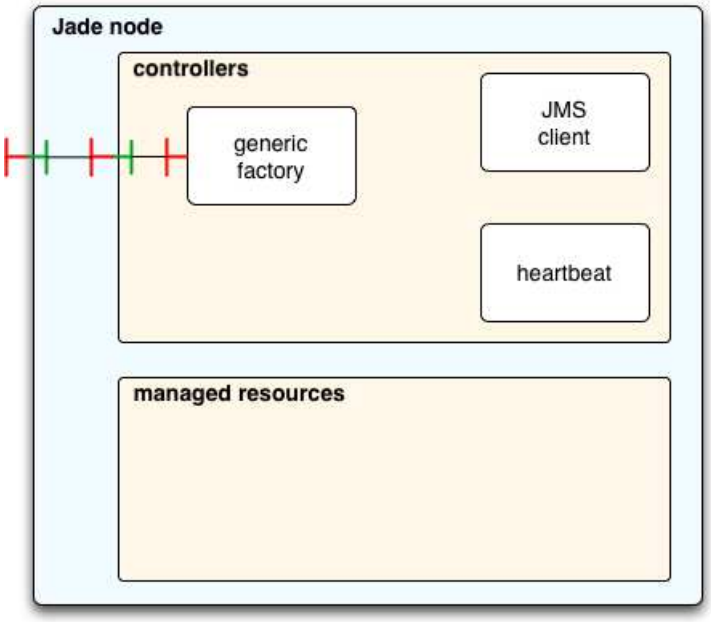


Figure 8.3: JADE Node

The **JMS client** is usually as simple as a JAR file and could be easily wrapped as a FRACTAL component. The local JMS client is used to support the asynchronous protocol for the autonomic discovery of JADE nodes. Each JADE node bootstrapping will publish on a *node discovery* topic that allows the self-deploy manager to autonomously discover newly available nodes.

The self-deployment manager is a replicated component, using an active replication that requires a deterministic implementation. Hence, it is mandatory that each correct replica of the self-deployment manager receives the same publish messages about bootstrapping JADE nodes. This can be achieved with JMS asynchronous messages using topics that are persistent and transactional senders. Each replica will react to the published bootstrap message in two steps:

- Establish a remote reference
- Update the System Representation

Establishing a remote reference to the bootstrapping JADE node is easily done through our component-aware Remote Method Invocation framework called now JADERMI. It suffices to add to the bootstrap publish message some very basic network information such as an IP address and a port number. With this simple information, a first pair of stub and skeleton can be created. This remote reference will be used by the causality layer between the *System Representation*.

But before causality can take place, the *System Representation* needs to be updated in order to reflect the presence of the newly bootstrapped JADE node. The *System Representation* is also replicated for the same fault-tolerance reasons, but it is co-located with the self-deployment manager replicas. Hence, each replica of the self-deployment manager will locally update the *System Representation*, bootstrapping the necessary knowledge for the causality layer that needs to apply reconfigurations done on the *System Representation* to the remote JADE nodes.

The **heartbeat generator** is for the failure detection of JADE nodes. It starts automatically when a JADE node bootstraps and starts multicasting heart beats to the replicas of the self-repair manager. The addresses of the replicas are provided by the self-deployment manager when establishing the remote reference to newly bootstrapped JADE nodes. When one of replicas of the self-repair manager fails, all the other replicas attempt to repair it on a newly allocated node. Of course only one repair order is executed. The address of the newly created self-repair manager replica is then sent to all the managed nodes by all of the self-repair managers. The nodes only process one instance of this message and update the information on self-repair replicas to which they broadcast heartbeats.

The **component factory** is an extended FRACTAL factory. As a regular factory, it supports the creation of FRACTAL components. In the JADE context, the factory is used remotely by the causality of the *System Representation* when it needs to request the remote creation of a component on a JADE node.

8.3 Modularity

A module is a loaded implementation, loaded from an on-disk physical package. In our Java prototype, a physical package is a JAR file and a module is implemented using a Java class loader that can load classes from the the JAR file.

In contrast to regular Java class loaders, modules are organized in a directed graph and not a hierarchy. The graph is the class loading delegation graph along the bindings between modules. Class loading delegation is the ability of a class loader to delegate the load of a class to another class loader. This is the core of our module layer that supports the cooperation between modules to load the classes requested by the Java Virtual Machine (JVM).

This cooperation is controlled through the bindings between modules, since modules are FRACTAL components. We decided to use the granularity of a Java package for our bindings. This means that for each exported Java package, a module provides a server interface, named with the very name of the exported Java package. Of course, the interface language is always the same, more on this Java interface below. Exported Java packages are not only named but versioned.

The resolution of modules is the creation of the bindings between imports and exports, respectively FRACTAL client interfaces and FRACTAL server interfaces of modules. This resolution is achieved by the deployment manager as it completes a reconfiguration plan. So in contrast to using an OSGI platform, a local JADE node behaves, regarding module bindings, as a slave to the master deployment manager.

8.3.1 Class loading

When a JVM needs a class, it needs it in the context of an already loaded class. For instance, it is loading that class or it is linking it lazily while executing the bytecode of a method. It does not matter, the point is that the JVM asks a class loader for the classes it needs. The class loader can define the requested class or it can delegate this process to another class loader. A newly created class belongs to the class loader that defines it, not the one that has been originally requested to load it.

To define a class, a class loader has to call back the JVM. In other words, the actual reification of the class object happens within the JVM, not within the class loader. The class loader needs the actual

contents of a class file to call back the JVM. It can find a class file on disk and load it in memory or it can generate directly in memory, it does not matter. The JVM expects a byte array formatted following the class file format specification, which is part of the JVM specification.

It is important to point out that class loaders are not part of the Java language specification. Class loaders are a runtime scoping of Java types. The type system of the Java language defines the concept of classes and interfaces. It defines the concept of Java packages, a hierarchical grouping of Java types, with explicit visibility rules based on the *public*, *protected*, and *private* attributes. But at compile time, type equivalence is based on name equivalence, assuming that only one class for a given name is visible on the compiler class path. Furthermore, the Java language has no concept of versions on types.

At runtime, class loaders do scope Java types however, but still don't use any concept of versions. The JVM no longer use name equivalence for comparing types but actually compares the class objects. Two Java types are equal at runtime if they are the same reified type, that is, the same class object (instances of the class *Class*). It is important however to realize that this means that if two class loaders load the same class file, this will create two class objects and therefore two Java types that are not equal although they have the name and they come from the same class file. It is important also to mention that class loaders do not support the reloading of classes.

8.3.2 Delegation

Delegation between class loaders determines which class loader defines a given class, it defines the overall coherence of the reification of Java types in a JVM. In JADE, it is driven by the bindings between modules. Each class loader, supporting one module, therefore delegates the decision to its module.

We have to take into account a limitation of the JVM, all types within the *java.lang* package cannot be reloaded by any class loader. Delegation must therefore happen to the system class loader that is created by the JVM when it starts. This system class loader in fact loads everything that is on the JVM classpath. In our approach, this JVM classpath does not contain any entries related to the different modules created by JADE.

For each such module, we create a class loader and we provide it the path to the one or more physical packages that it needs. This is simply done through a URL class loaders that takes as its classpath a set of URLs. These URLs are local ones pointing within the local repository for physical packages that is present on each JADE node. It is important to point out though that we modified this URL class loader so that it delegates to its module when requested a class by the JVM. Indeed, we don't want to use the original implementation that implements the hierarchical design of class loaders, delegating to a parent class loader.

Modules have a delegation policy that corresponds to a graph, corresponding to the graph of bindings between modules. Before anything else, a module attempts to load the class through the system class loader, so to respect the aforementioned limitation. If the system class loader does not find the requested class, the module can either attempt to find it locally (in its physical package) or use its bindings, representing imported Java packages. It first searches through its imports, checking the package names. If no import is found for the package of the requested class, the class is searched for locally, within the physical packages of the module. Of course, this is a recursive process since when

searching an import, one delegates the request for a class to the exporter module that simply repeats the same delegation policy.

It is very important that it uses its imports first. The rationale is the overall consistency of the module layer, which will be discussed in the next section. Simply put here, we want imports to dominate the local packages so that the global resolution decided by the self-deployment manager can force the hiding of a local package if necessary.

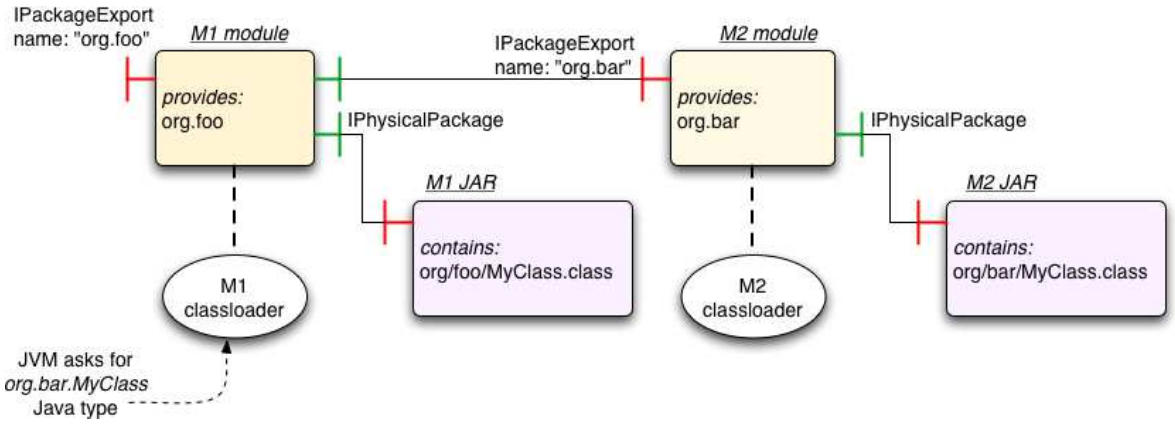


Figure 8.4: Modules and physical packages.

This overall scheme is depicted in Figure 8.4. It shows two modules, assembled through bindings for the packages *org.foo* and *org.bar*. When the class loader of the module M1 is requested a class *org.bar.MyClass*, it upcalls its module. The module M1 first extracts the package name from the fully qualified class name, *org.bar* in this case. If finds that it has an import for that package, an import bound to the module M2. The request for the class *MyClass* is forwarded to the module M2, that repeats the process. However, the module M2 does not have an import for the package *org.bar*, hence it is searched locally. The module M2 requests its class loader to load the class from its associated JAR file (physical package). If it is found, the class loader of M2 defines the class in the JVM and returns the class object. If it is not found, the class loader raises a *ClassNotFound* exception.

The delegation that occurs through bindings relies on the functional services exported by modules. As already mentioned, modules are regular JADE components and are therefore organized around a service-oriented paradigm, even though the functionality of these services is to load classes, something perceived as arcane details of a Java Runtime Environment. Each exported Java package corresponds to the export of a FRACTAL server interface with the interface language given in Figure 8.5. IPackageExport interfaces expose the classes loaded by a module to other modules via the *loadClass* method. Other resources contained within the modules, such as image files, can be accessed through the *getResource* method.

In the FRACTAL model, server interfaces not only have a language interface but they are also identified by a name. The bindings are actually resolved on that name and not on the language type of the interface. Therefore, the name of an exported package must be the very name of that exported package. For instance, if a package *org.foo* is exported by a module, it is done so through an exported server interface whose name is *org.foo*. Furthermore, the version of that package needs to be added,

```
public interface IPackageExport {

    public Class loadClass(String className) throws ClassNotFoundException;
    public URL getResource(String resourceName);
    public InputStream getResourceAsStream(String resourceName);

}
```

Figure 8.5: The Java signature of the *IPackageExport* interface

so the name is in reality *org.foo#1.4.3*. Hence, each export has a specific name but they all share the same language type, the *IPackageExport* interface.

8.3.3 Module resolution

The process of *resolution* of module components consists in binding its imports to exports from other modules. These imports and exports are meta-declared for each module, using a domain specific language, illustrated in Figure 8.3.3.

```
<modules>

  <module id="client" version="1.0.0">
        <import package="itfs" version="1.0.0"/>
        <import package="system" version="1.0.0"/>
        <content file="client.jar"/>
  </module>

  <module id="server" version="1.0.0">
        <export package="itfs" version="1.0.0"/>
        <import package="system" version="1.0.0"/>
        <content file="server.jar"/>
  </module>

</modules>
```

Figure 8.6: A sample module repository description file

This file describes two modules—the client and the server one. The server module contains, within the *server.jar* file, the resources needed by the application components created from this module to work. It also contains the Java interface over which other components can be bound to those application components. This interface is defined in the *itfs* package. The client module, contained within the *client.jar* file, needs this Java package in order to be resolved.

When the above file is parsed by the component framework underlying JADE, the *client* and *server* module components (both in version 1.0.0) are instantiated and added as subcomponents to the composite called the *module repository*. There is only one such composite per JADE node, each running in a single instance of the JVM. The purpose of *module repository* composite is to permit the bindings between imports and exports of the various modules to happen while respecting the FRACTAL model. Indeed, the FRACTAL model imposes that a binding can be established solely through sibling components, sharing the same parent composite.

These bindings are created by the self-deployment manager when it completes a reconfiguration plan. It corresponds to solving the constraint system modeled as imports and exports on a know set of modules. Of course, imports must be matched to corresponding exports, using the package names and versions. An export is identified by a package name and its version; an import is expressed via a package name and a range of compatible versions. This scheme introduces a great flexibility that needs close attention.

First, the presence of a range of compatible versions introduces a flexibility in the resolution process. The resolver (part of the self-deployment manager) may have a choice between different versions exported by different modules. Indeed, a module may export only one version of a package but different modules may export different versions of the same package. As we explained in Section 6.3.3, we chose to retain a unique version and avoid the complex issues related to multiple versions.

Second, we need to consider the overall consistency of the module assembly from the point of view of the Java type system. It is paramount that we avoid multiple modules to define the same Java type. Delegation is the raw mechanism for avoiding this, as we already explained. But it is not sufficient, the problem has to be handled at the resolver level. A correct solution requires that exports are implicit imports. Indeed, if we have two modules that export the same package P, the resolver will retain only one for the importers of P, but what about the two exporters? Without imports, they will both load and define the Java types of the package P. This is obviously incorrect since we have two reified class objects for the same Java type.

With exports being implicit imports, the overall assembly of modules behave correctly. For each exported package, the resolver retains one specific exporter of that package. The resolver will then bind all imports for that package (including the implicit ones) to that specific exporter. This means that any exporter of that package will know, through its implicit import, who has been chosen to export the package. If it is itself, it loads from its local package, otherwise it delegates.

Implicit imports can be made explicit. The reason is that an exporter of a Java package exports a specific version of that package but it may be able to accept other versions, lower or higher. For instance, a module may export a package P in version 1.4 but be able to accept any 1.x version. Hence, a module can define an explicit import with a version range for any of its exports. This provides greater flexibility for the resolver to bind modules, thereby allowing greater number of various assemblies to be resolved.

8.3.4 Module updates

An interesting question about Java-based modules arises in the context of dynamic updates of modules. Is there any superior organization of modules regarding the separation between exported Java packages and those private encapsulated ones? The pursued idea here would be to allow updating the private packages while minimizing the transitive closure of modules that need to be stopped because they use the exported public packages.

In other words, we would like to be able to update the implementation of a functional component with minimal impacts on the availability of the overall system. In all cases, functional components that use exported services from components that need to be updated will have to be stopped. This is unavoidable. What is avoidable is to force them to be not only stopped but disposed of. Indeed, a revoked functional service requires the client component to be stopped. A revoked exported Java

package requires that any module importing that package be non only stopped but also disposed of. This is because Java does not support relinking loaded classes dynamically. Hence, the class loader of a module has to be thrown away and a new one created every time one binding of that module is changed. The disposal of the module cascades over all its functional components since such components cannot exist without their implementation.

As one can see, updating modules could be a very expensive operations. Not only can it be expensive from the actual amount of work to be done for disposing of modules and reloading them, but it is also expensive in terms of the entailed interruption of service that end users are likely to notice. Hence, having a module per component that captures not only the encapsulated private implementation but also the public exported types does not seem optimal. Requesting developers to handle this separation of concerns is however a serious burden, unlikely to be accepted and furthermore likely to be error-prone.

Fortunately, we can propose an automated approach to this optimization. JADE runtime can automatically separate the two facets of a module in two different class loaders. If, for example, we want to bind two components *Client* and *Server* using the interface *ServiceItf*, the *ServiceItf* has to be loaded by the module common to components *Client* and *Server*. At the same time, the *Client* and *Server* components' implementation classes must be loaded by independent modules to allow for code isolation and versioning of the two components' implementations. The preliminary version of this approach has been described in (Kornaś et al. 2004).

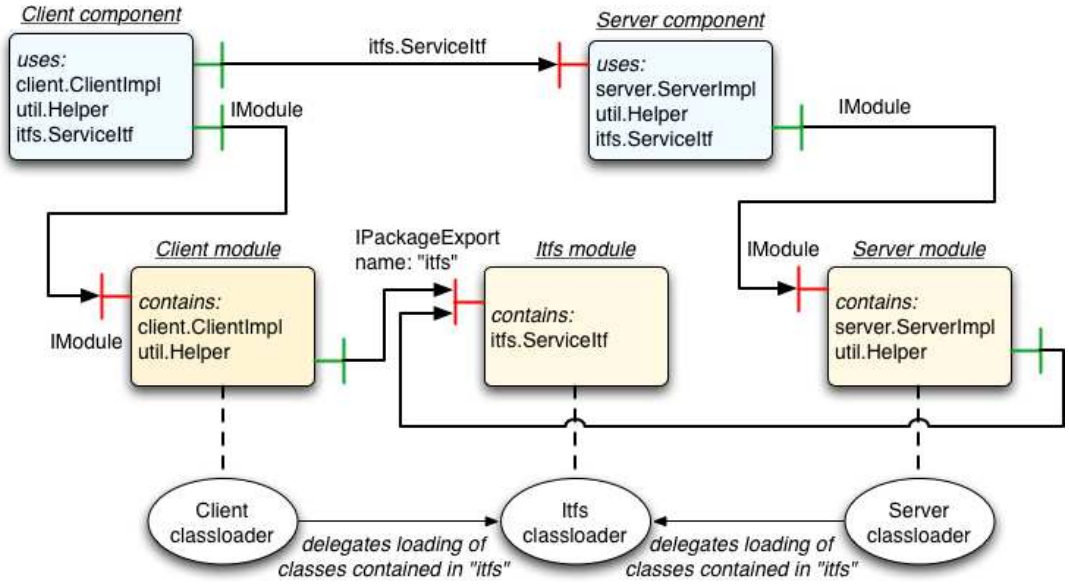


Figure 8.7: Organization of optimized modules and their class loaders.

Figure 8.7 presents the resulting organization of the modules of our simple client-server application depicted in figure 8.8. Classes building the application are separated into the ones that are exchanged between components and the ones that components use privately. This leads to the use of three distinct modules: *client* and *server* modules load the components' private classes, whereas the *itfs* module loads the interface via which the two components are connected.

Presented solution allows flexible updates of the server component's implementations without the

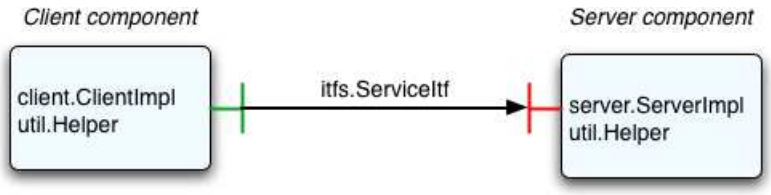


Figure 8.8: A simple client-server component application in Java.

impact on other components running in the system. Such capability is important in long-term maintenance of component-based applications. Note that after an implementation of a given component is updated, as long as the old version of a component is no longer used in the system, the module that loaded the old version of a component can be garbage-collected to decrease the memory footprint of the application.

Updates of component's interfaces are also possible. However, since interfaces are referenced by the components' implementation classes, the update of an interface for a component *C* requires the update of all the components bound to *C* through this interface. This is depicted in figure 8.9: interface *I* is used to bind components *C2* to *C3*. Interface *J* binds the component *C1* to *C2*. Updating *I* requires an update of both *C2* and *C3* because both of these components use this interface. However, it does not require an update of *C1*, even though it is bound to *C2*. This is due to the fact that the *J* interface creates a "reconfiguration border". Therefore, after the interface *I* has been replaced by the new versions for components *C2* and *C3*, components *C1* and *C2* can continue using the "old" version of *J* and the implementation of component *C1* does not need to be reloaded.

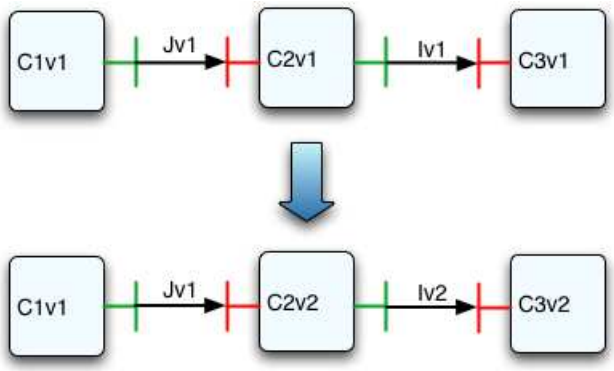


Figure 8.9: Interface reconfiguration example.

8.4 Physical Packages

Physical packages in this implementation of JADE are JAR files. Each JADE node implements a local repository for downloaded physical packages. A local repository acts as a local cache, therefore avoiding multiple expensive downloads of physical packages. Furthermore, the local repository participates in the overall fault-tolerance of JADE as it stores multiple copies of the same JAR files across several nodes. This prevents node failures from making some JAR files unavailable.

At bootstrap, when a Java node is started, it has to reify its local repository of physical packages. Indeed, physical packages are modelled as JADE components and the local repository as a composite. The bootstrap process therefore creates the *repository* composite and adds a *physical package* subcomponent for each locally stored JAR file. Once a JADE node will be discovered by JADE Boot (through our heartbeat detectors), the JADE node will upload the description of the contents of its local repository so that JADE Boot knows where to find copies of JAR files.

8.5 Garbage Collection

Garbage collection is applied to both deployed modules and physical packages, on each JADE node. The rationale for an approach based on automated garbage collection is that it provides greater fault tolerance through a simpler design. Indeed, deployed physical packages and the modules created from these physical packages do not need special care when rollbacking atomic reconfigurations of the architecture; they will be automatically garbage collected if no longer used locally.

This preserves most autonomic managers from having to know about modules and physical packages; they can focus on the functional assembly of components and let the deployment manager complete the reconfiguration plans they produce as explained in Section 6.4. Nevertheless, such plans need to have explicit undeploy tasks for functional components as this corresponds to an explicit reconfiguration of the reflexive architecture. No garbage collector can automatically decide this since a functional component is always reachable within the reflexive architecture.

At the very least it is part of the composite that models the software feature that component is part of. For instance, a Servlet engine is part of a Web Application Server, from a feature perspective. This means that a Web Application Server has the potential functionality of hosting servlets. Now, each actual instance of a Web Application Server is represented with another composite that has a servlet engine or not. This corresponds to the two views: one is the abstract software description in terms of software features while the other is the actual configuration of a given instance of that software.

Although modules and physical packages are regular components and as such always reachable within the reflexive architecture, JADE runtime has enough semantics to be able to apply an automated garbage collection. Indeed, modules and physical packages are downloaded onto a given JADE node to support the instantiation of functional components. If no functional components need a particular module, it can be garbage collected. If no particular module needs a physical package, it can also be garbage collected.

It is important that our garbage collection be relatively aggressive regarding the reclamation of modules. Indeed, modules are loaded implementations and thereby waste main memory, potentially important amounts for large modules encapsulating many classes. This is especially important because once loaded, Java classes are not individually unloaded; only an entire class loader can be unloaded. This is only possible if there are no Java references to the class loader itself, to any of the class it loaded, and to any instance of these classes. These invariants are true in JADE when a module is no longer used by any functional components.

The reclamation of physical packages has to be tempered because of our design of the local repository that participates to the overall fault tolerance of JADE. The automated garbage collector will discover that physical packages are no longer needed locally, but it will not actually reclaim the phys-

ical packages. It will inform the local repository that certain physical packages are no longer needed, the repository will reclaim the physical packages only if enough other copies are available on other nodes.

8.6 Conclusion

In this chapter, we presented the deployment-related new design of the JADE platform. Although, this new design is still on-going work at the time of this writing, still facing open research challenges, we reported here the design of a significant first step towards self-deployment which in turns helps with self-repair.

We have most of the raw mechanisms in place for autonomic self-deployment. We have an autonomic self-discovery of nodes. We have the ability to download, cache, and reclaim physical packages, which participates to the overall fault-tolerance as our scheme supports replicated storage of physical packages. We have a modularity layer that can load component implementations from physical packages.

We have a powerful resolution scheme for our modules. We believe that we stroke the right balance between expressive power and understandability by developers. This balance is in between the Release 3 and 4 of the OSGI modularity specification. The Release 3 retained only one version of each exported package, but the versioning scheme was the official Java versioning specification that prevents any incompatible evolutions of Java types; not something that we consider practical. The versioning scheme has changed in Release 4, we adopted the same one.

However, we did not retain some of the more arcane support for modularity, mostly adopted for supporting the *bundle-izing* of legacy Java code[1]. We did not retain split packages. We did not retain the inflexible requirements at a module level. We did not retain multiple versions, as we discussed in 6.3.3.

We are experimenting with optimizations around modularity. We presented one optimization around the support of updates of modules. The core idea is to recognize that many updates are about incremental fixes, usually limited to fixing the implementation and not changing the exported types. We propose a two-classloader scheme that prevents much of the overhead of such updates.

Our design is compatible with the needs of our autonomic self-repair goal. Our self-deployment manager is co-replicated with the *System Representation*, providing fault-tolerance to the failures of JADE node. The cardinality of the replication is kept by the self-repair manager that detects such failures and repairs the failed JADE node, creating a new node identical to the one that failed.

We are missing the support for the atomicity of the reconfigurations. The atomicity of the membrane level operations are easier to obtain since any controller operation has a reverse. Traditional roll-back techniques can therefore be applied. Regarding operations on functional interfaces, the solution is much harder without any virtual machine support. This is because most of the operations exposed by components through their functional interfaces do not have a reverse. Relying on programming rules is a burden on developers and extremely error-prone. Recovery touches about 80%

[1]Personal communication with Olivier Gruber, ex-expert to the OSGI Alliance in the Core Expert Group that specifies the core OSGI framework.

of the code of a database system, it would be unrealistic to expect component developers to succeed in providing atomicity-reliable code.

We are missing tools. Indeed, the compilation of modules written in Java has to happen with the same visibility rules as the actual execution of the same modules. This requires that the Java compiler uses the same modularity model to scope types when compiling. For instance, the encapsulation of private types must be respected or the compiler may see multiple types with the same name. Also, the visibility at the module level ensures the use of proper versions of types, something that current compilers are unaware of.

Beyond compiling, tools are also necessary for the management of the complete lifecycle of our components. Developers need to decide how to package their Java packages in modules. They have to version both modules and Java packages. They further need to bundle modules in physical packages. The physical packages have to be versioned and published to public repositories. These are complex tasks, poorly supported by existing traditional tools. As we mentioned, Eclipse started to include some of this tool support, but it is OSGI specific.

Résumé de chaptire 9

Dans le chapitre 9 de cette thèse on résume nos travaux sur le déploiement dans le contexte des systèmes autonomes pour la gestion des applications a composants base sur l'architecture. D'abord, dans la section 9.1 on résume brièvement les problèmes adressé dans cette thèse. Ensuite, dans la section on résume nos contributions. Puis, dans section on illustre les pistes pour nos travaux futurs.

Chapter 9

Conclusions and Future Work

Contents

In this final chapter of the thesis we summarize our work on deployment in the context of autonomically managed component-based software architectures. We begin by briefly reminding the problems that we have investigated in Section 9.1. Then we present a summary of our contributions. Finally, we describe the areas for future investigation.

9.1 Problems addressed

Due to the growing size of software projects, the software development process became increasingly complex in the past years. As a result, a component-based approach imposed itself as a *de facto* standard in software production. This is because a component based approach provides better software engineering through modularized development. Even though the component-based approach proves effective in addressing the development challenges of modern software, it shows limitations in terms of runtime software management.

Through the analysis of related work performed in the first part of this thesis, we showed that these limitations are a consequence of ignoring deployment-related aspects of software management. Most existing component frameworks assume that implementations of software components are always available on target machines. As a consequence, they do not investigate the issues of dependencies between those implementations as well as their isolation and versioning. The lack of proper mechanisms for the management of component implementations has major implications on the construction of autonomic software systems based on the explicit architecture provided by components. One of such systems is JADE—the context of our work. In its initial version JADE, like other frameworks described in the analysis of the state of the art, did not consider deployment as an integral part of software management. One would expect an autonomic management framework to be able to perform

all the deployment activities described in Section 1.2, such as the installation, dynamic updates and removal of software components. Unfortunately, this is not the case. Our work proposed and validated a solution that, although designed for JADE, is applicable to most component-based deployment systems.

9.2 Review of principal contributions

Chronologically, the first contribution of this thesis is an architecture-based deployment system for the JADE autonomic management framework. This deployment system addresses several limitations of the existing solutions described in the analysis of the state of the art. It provides the just-in-time installation of software components managed by JADE. It also allows for runtime isolation and versioning of these components. Finally, it supports in certain cases the dynamic updates of the implementations of components.

This deployment system reuses OSGI as a packaging and modularity mechanism and FRACTAL as a component model. The usage of OSGI was motivated by the completeness of this framework in terms of providing a development and deployment environment for Java software. Through various real-world examples described in this thesis, we illustrated the applicability of our approach to the deployment of industrial grade component-based middleware, such as the JOnAS Java EE server and the Joram JMS server. This version of our deployment system for JADE is successfully used within the ObjectWeb JASMINE project, where it supports ADL-based deployment of complex software.

However, when trying to apply this first version of the deployment system to JADE itself, we have realized that the combination of OSGI and FRACTAL is incompatible with the architecture-based approach to software management. The principal issue arises from the fact that OSGI takes its own autonomic decisions which concern certain aspects of the software architecture. Namely, dependencies between implementations of components are resolved internally by OSGI, which is not visible to JADE. As a result, JADE's autonomic managers can potentially take decisions about software architecture being deployed which are incompatible with the ones taken by OSGI.

Therefore, the second contribution of this thesis is the integration of the deployment-related information with the component model. This integration is performed through several extensions to the basic component model (FRACTAL in our case). The first extension is about modules and physical packages. Both are represented as software components within the explicit software architecture. However, the roles of these components differ from the roles of regular "business" components. The role of modules is to represent the loaded implementations of components. This is important, because component frameworks are usually implemented in typed programming languages, in which the resolution of language types has an impact on the correctness of the whole software architecture. The role of physical packages is to represent the on-disk implementations of components. This models the local installation of software artefacts, such as compiled code.

The second extension is what we call the resolver. A resolver is an additional autonomic manager, part of the explicit architecture of JADE, which decides how modules are bound to one another and how runtime components are created from these modules. To establish a correct software architecture, the resolver manager cooperates with other autonomic managers, such as the self-reparation and self-

optimization ones. It does so by being the one who completes the raw deployment plan, enhancing it with the deployment-related information. This makes the management process easier and more abstract for other autonomic managers within the JADE system.

Introduction of modules, physical packages and the phase of resolution of their dependencies has an impact on the lifecycle of components. We have handled this impact by extending the basic start/stop life cycle automaton provided by the FRACTAL model with the resolution and destruction phases. No application components can be created unless their corresponding module components are resolved. Whenever the resolver decides that a given module component becomes unresolved, the application components created from this module need to be destroyed.

To prove the correctness of our approach, we have built a prototype implementation of the deployment environment for JADE components in Java. This second implementation built during this thesis does not reuse OSGI as the installation and versioning layer. Instead, it is built directly on top of Java class loaders. The advantage of this implementation is that the implementations of software components are part of the explicit software architecture managed by JADE and the decision on how these implementations are resolved is taken by JADE internally. Thus, JADE framework can take correct decisions when reconfiguring the software architecture that it manages.

9.3 Future work

Our research work during this thesis focused essentially on providing models and a core infrastructure for integrating deployment into an architecture-based approach to software management. This research opened several interesting areas for future investigation around the deployment of component-based software.

9.3.1 Distributed deployment

Even though we have identified the distributed execution of the deployment plan as a core aspect of software deployment, we have not thoroughly investigated this issue. Instead, we have mainly focused on the local installation and versioning of software components. It would therefore be interesting to couple the approach described in this thesis with a sophisticated distributed deployment plan. One possible approach would be to replace the current deployment plan used within JADE by a software program written directly in a concurrent, distributed language, such as Oz (Smolka 1995) or Erlang (Armstrong et al. 1996). This would allow for building complex deployment workflows.

9.3.2 Atomic reconfigurations

Currently, neither of the two prototypes that we have built for this thesis support fully atomic reconfigurations of the software architectures. At present, the only atomicity that JADE offers is the one resulting from the reversibility of membrane operations provided by the FRACTAL model. On top of this simple mechanism, roll-back techniques can potentially be built. However, in order to obtain a full support for atomic operations on the level of functional interfaces, one would need to have this atomicity provided by the virtual machine.

9.3.3 Optimizations around modularity

The subject of optimizations of module components also requires further attention. In Section 8.3.4 we presented an example of how the organization of modules can be optimized in order to improve the handling of dynamic updates of components' implementations. The presented approach limits the impact of updating a component on other components. This, however, is only one of many possible optimizations. It would be interesting to evaluate the trade-offs between various policies in the creation of modules.

9.3.4 Tooling

There is a lot of possible work to be done in terms of tools for the development and deployment of components and their modules. Indeed, when the code of components written in Java is compiled, the compilation should be performed with the same visibility rules as the actual execution of these components. This requires proper tools that are capable of applying the same modularity model for scoping Java types at compilation time as the visibility model used by the component runtime. In practice, this means that not only the encapsulation rules of private types should be respected at compilation, but also that the compiler should be able to see and handle multiple versions of types with the same name. This is something that existing compilers are not supporting, even though certain tools, such as the Eclipse Plug-In Development Environment (PDE), are heading in that direction.

Tools are also needed for managing the complete deployment life cycle of software components. These tools would allow the developers to decide how they package their components into versioned artefacts for the release phase. Again, Eclipse provides some of these functionalities, but it is currently incomplete and OSGI specific.

Beyond compiling, tools are also necessary for the management of the complete lifecycle of our components. Developers need to decide how to package their Java code in modules. They have to version both modules and Java packages. They further need to bundle modules in physical packages. The physical packages have to be versioned and published to public repositories. These are complex tasks, poorly supported by existing traditional tools. As we mentioned, Eclipse started to include some of this tool support, but it is OSGI specific.

Conclusion de thèse

A cause de l'augmentation de la taille des projets logiciels modernes, le processus de développement est devenu très complexe ces dernières années. Par conséquence, un approche a base des composants s'est impose comme un *de facto* standard dans la production des logiciels. C'est du au fait que cette approche fourni une meilleure ingénierie des logiciels grce au développement modulaire. Hélas, cette approche a des limitations en terme de la gestion des logiciels a l'exécution.

A travers d'analyse des travaux connexe décrit dans la première partie de cette thèse, on a prouvé que ces limitations sont une conséquence de du fait que les aspects lie au déploiement sont souvent ignoré dans la domaine de gestion des logiciels. La plupart des plateformes pour la gestion des composants simplement assume que les implémentations des composants sont toujours disponible sur les machines cibles. Par conséquence, ces plateformes ne s'intéressent pas aux problèmes des dépendances entre ces implémentations ainsi que leur isolation et versionnement. La manque des mécanismes de gestion des implémentations des composants a un impact important sur la construction des systèmes autonomes qui se basent sur une architecture explicite fourni par les composants. Un de ces systèmes est JADE—le contexte de notre travail. Dans sa version initiale JADE, comme la plupart d'autres plateformes décrit dans l'état de l'art, n'a pas considéré le déploiement comme une partie intégrale de la gestion des logiciels. Pourtant, une plateforme de gestion des logiciels devrait supporter toutes les opérations de gestion décrit dans la section 1.2, tel que l'installation, les mises a jour dynamiques et la suppression des composants. Malheureusement, c'est pas le cas dans JADE. Nos travaux ont propose et valide une solution qui, même si conu pour JADE, est applicable a la plupart des systèmes de déploiement pour les applications a base des composants.

La première contribution de cette thèse est un système de déploiement base sur l'architecture pour la plateforme de gestion autonome JADE. Ce système résout plusieurs limitations des autres solutions existants décrit dans l'état de l'art. Il fournit également la capacité d'installation a la volé des composants géré par JADE, supporte l'isolation et versionnement des composants et, dans certains cas, le mise a jour dynamique de ces composants. Ce système utilise OSGI comme solution pour le paquetage et modularité et un modèle a composants FRACTAL pour l'architecture explicite des logiciels. Cette version de système de déploiement pour JADE était applique avec succès dans le contexte industriel pour déployer des intergiciels tel que le serveur Java EE JOnAS et le serveur JMS Joram dans le cadre de projet JASMINE ObjectWeb.

Hélas, ce système ne peut pas être applique au JADE lui-même a cause des tensions entre OSGI et FRACTAL. En effet, OSGI prend ses propres décisions qui ont un impacte sur la validité de l'architecture logiciel, tel que la résolution des dépendances entre les implémentations des composants. Ces décisions ne sont pas visibles pour les gestionnaires JADE. Par conséquence, les gestionnaires autonome de JADE peuvent potentiellement prendre des décisions concernant l'architecture géré qui ne sont pas compatibles avec les décisions pris par OSGI.

La deuxième contribution de cette thèse est l'intégration d'information lié au déploiement avec le modèle a composants. Cette intégration est effectué a travers des plusieurs extensions de modèle,

tel que la représentation explicite des modules et des paquetages physiques en forme des composants. Les rles des ces composants sont différents que les rles des composants “réguliers”. Les modules représentent les implémentations des composants chargé par le système. C’est important car les plate-formes a composants sont pour la plupart implémenté dans les langages de programmation typé, dans lesquelles la résolution des types langage a un impacte sur la validité des architectures logiciels. Les composants paquetages physiques représentent le code installé sur les machines cibles. L’introduction des ces deux notions dans le modèle a composant implique l’introduction d’un nouveau gestionnaire autonome dans JADE—le *resolver*. Ce gestionnaire décide comment les modules sont résolu entre eux et comment les composants “applicatives” sont crées a partir des modules. Pour établir une architecture logicielle correcte, le resolver coopère avec les autres gestionnaires autonomes de JADE a travers d’un plan de déploiement, ce qui rends le processus de gestion plus abstrait pour les gestionnaires autonomes de JADE. Enfin, l’introduction de toutes ces éléments a un impact sur le cycle de vie des composants. Notamment, les composants applicatives peuvent être crées que si leurs modules correspondantes sont résolu. également, si le resolver décide qu’un module devient non-résolu, les composants applicatives crées a partir de ce module doivent être détruits.

On a validé ce deuxième approche au déploiement avec un nouveau prototype pour JADE. Cette deuxième implémentation ne utilise pas OSGI mais fourni son propre système d’isolation et résolution des types langage, basé sur les chargeurs des classes Java (les *class loaders*). L’avantage de cette version de JADE est que les implémentations des composants font partie intégrale de l’architecture logicielle géré par JADE et donc toutes les décisions sur la reconfiguration d’architecture sont prises par JADE lui-même.

Dans les travaux futurs qui suivront cette thèse, on pense se focaliser sur plusieurs aspects de déploiement. D’abord on aimerai se focaliser sur le plan de déploiement. Dans la version actuelle, le plan est simpliste en terme d’exécution repartie ainsi que la synchronisation des taches. Ensuite, les aspects de atomicité des reconfigurations nécessitent plus de travail. C’est également le cas pour la optimisation de modularité. Enfin, la partie outillage pour le développement et déploiement des composants n’est pas a l’heure actuelle suffisamment complète.

Bibliography

Abdellatif, T.: 2005, Enhancing the Management of a J2EE Application Server using a Component-Based Architecture, *Proceeding of the 31st EUROMICRO Conference (EUROMICRO'2005)*, Porto, Portugal.

Abdellatif, T., Korna´s, J., and Stefani, J.-B.: 2007, Reengineering j2ee servers for automated management in distributed environments, *IEEE Distributed Systems Online Journal, vol. 8 no. 11*.

Abdellatif, T., Kornas, J. and Stefani, J.-B.: 2005, J2ee packaging, deployment and reconfiguration using a general component model, *Component Deployment*, pp. 134–148.

Armstrong, J., Virding, R., Wikstr¨om, C. and Williams, M.: 1996, *Concurrent Programming in Erlang*, 2nd edn, Prentice Hall.

Bailey, E. C.: 2000, Maximum RPM, *http://www.redhat.com/docs/books/max-rpm/max-rpm-html/index.html* .

Bailliez, S.: 2005, Class loader architecture in Geronimo. http://marc.theaimsgroup.com/?l=geronimo-dev&m=111876562108164.

Bouchenak, S., Boyer, F., Krakowiak, S., Hagimont, D., Mos, A., Stefani, J.-B., de Palma, N. and Quema, V.: 2005, Architecture-based autonomous repair management: An application to j2ee clusters, *srds* **0**, 13–24.

Bruneton, E., Coupaye, T., Leclercq, M., Qu´ema, V. and Stefani, J.-B.: 2004, An Open Component Model and its Support in Java, *Proceedings of the International Symposium on Component-based Software Engineering (CBSE'2004)*, Edinburgh, Scotland.

Candea, G., Kiciman, E., Zhang, S., Keyani, P. and Fox, A.: 2003, Jagr: An autonomous self-recovering application server., *5th Annual International Workshop on Active Middleware Services (AMS 2003), Autonomic Computing Workshop*.

Cheng, S.-W., Huang, A.-C., Garlan, D., Schmerl, B. R. and Steenkiste, P.: 2004, Rainbow: Architecture-based self-adaptation with reusable infrastructure., *1st International Conference on Autonomic Computing (ICAC 2004)*.

Community, T. S.: 2004, The Spring framework. http://www.springframework.org/.

Corwin, J., Bacon, D. F., Grove, D. and Murthy, C.: 2003, Mj: a rational module system for java and its applications., *Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'2003)*, Anaheim, California, USA.

Deng, G., Balasubramanian, J., Otte, W., Schmidt, D. and Gokhale, A.: 2005, Dance: A qos-enabled component deployment and configuration engine, *3rd Working Conference on Component Deployment (CD 2005)*, Grenoble, France.

Desertot, M., Donsez, D. and Lalanda, P.: 2006, A dynamic service-oriented implementation for java ee servers, *IEEE SCC*, pp. 159–166.

Dolstra, E., de Jonge, M. and Visser, E.: 2004, Nix: A safe and policy-free system for software deployment, *LISA '04: Proceedings of the 18th USENIX conference on System administration*, USENIX Association, Berkeley, CA, USA, pp. 79–92.

Exertier, F.: 2004, J2ee deployment: The jonas case study, *CoRR* **cs.NI/0411054**.

Fleury, M. and Reverbel, F.: 2003, The jboss extensible server, *International Middleware Conference* .

Flissi, A. and Merle, P.: 2006, A generic deployment framework for grid computing and distributed applications, *2nd International OTM Symposium on Grid computing, high-performAnce and Distributed Applications (GADA'06)*, Vol. 4279 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1402–1411.

Formalized Class Loading: 2007.

Fractal Deployment Framework: 2006.

Framework, S.: 2006, Spring Dynamic Modules for OSGi. http://www.springframework.org/osgi.

Ganek, A. and Corbi, T.: 2003, The dawning of the autonomic computing era, *IBM Syst. J* **42**(1), 5–18.

Goldsack, P., Guijarro, J., Lain, A., Mecheneau, G., Murray, P. and Toft, P.: 2003, SmartFrog: Configuration and Automatic Ignition of Distributed Applications.

Grid Computing Info Centre: 2002. `http://www.gridcomputing.com`.

Group, O. M.: 1996, Common Object Request Broker Architecture (CORBA). http://www.corba.org.

Group, O. M.: 2006, *Deployment and Configuration of Component-based Distributed Applications Specification*, OMG Document, formal/2006-04-02.

Hall, R. S.: 2004, A Policy-Driven Class Loader to Support Deployment in Extensible Frameworks, *Proceedings of the International Conference on Component Deployment (CD'2004)*, Edinburgh, Scotland.

Hnetynka, P.: 2005, Making deployment process of distributed component-based software unified, *PhD thesis*, Charles University, Prague.

J2EE Deployment Specification (JSR88): 2005.

J2EE: Java 2 Platform, Enterprise Edition: 2002.

J2EE Management Specification (JSR77): 2005. `http://jcp.org/jsr/detail?id=77.jsp`.

Java Message Service Specification Final Release 1.1: 2002. Sun Microsystems, http://java.sun.com/products/jms/docs.html.

Java Module System (JSR277): 2004. http://www.jcp.org/en/jsr/detail?id=277.

JOnAS: Java Open Application Server: 2005.

JORAM: Java Open Reliable Asynchronous Messaging: 2002. Objectweb, `http://www.objectweb.org/joram/`.

Julia: Fractal Composition Framework Reference Implementation, Objectweb: 2002. http://www.objectweb.org/fractal.

Kephart, J. O. and Chess, D. M.: 2003, The vision of autonomic computing, *IEEE Computer* **36**(1), 41–50.

Korna´s, J., Leclercq, M., Qu´ema, V. and Stefani, J.-B.: 2004, Support pour la reconfiguration d'implantation dans les applications a composants java, *CoRR* **cs.NI/0411082**.

Luer, C. and van der Hoek, A.: 2004, JPloy: User-Centric Deployment Support in a Component Platform, *Proceedings of the 2nd International Working Conference on Component Deployment (CD'2004)*, Edinburg, Scotland.

Open Services Gateway Initiative, OSGi service gateway specification, Release 4: 2005.

Oppenheimer, D., Ganapathi, A. and Patterson, D.: 2003, Why do Internet services fail, and what can be done about it?, *4th Symposium on Internet Technologies and Systems (USITS 2003)*.

Qu´ema, V., Balter, R., Bellissard, L., F´eliot, D., Freyssinet, A. and Lacourte, S.: 2004, Asynchronous, hierarchical, and scalable deployment of component-based applications, *Component Deployment*, pp. 50–64.

Sheng, L. and Bracha, G.: 1998, Dynamic class loading in the java virtual machine, *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'1998)*, Vancouver, Canada.

Smolka, G.: 1995, The Oz programming model, *in* J. van Leeuwen (ed.), *Computer Science Today*, Springer-Verlag LNCS 1000, Berlin, pp. 324–343.
URL: *http://www.mozart-oz.org/papers/abstracts/volume1000.html*

Strnisa, R., Sewell, P. and Parkinson, M. J.: 2007, The java module system: core design and semantic defi nition, *OOPSLA*, pp. 499–514.

Subramonian, V., Deng, G., Gill, C., J.Balasubramanian, Shen, L., Otte, W., Schmidt, D., Gokhale, A. and Wang, N.: 2007, The design and performance of component middleware for qos-enabled deployment and confi guration of dre systems, *Journal of Systems and Software* **80**(5).

Sun Microsystems Inc.: 1996, Rmi - remote method invocation.

Szyperski, C.: 1998, *Component Software: Beyond Object-Oriented Programming*, ACM Press and Addison–Wesley, New York, NY.

The Apache Software Foundation: 2005. `http://apache.org`.

Understanding J2EE Application Server Class Loading Architectures: 2002. The Server Side, http://www.theserverside.com/.

WebLogic Server, Application Class loading: 2004. `http://www.bea.com/`.

WebSphere Software Information Center, Class loaders: 2003. `http://www.ibm.com/`.

White, S. R., Hanson, J. E., Whalley, I., Chess, D. M. and Kephart, J. O.: 2004, An architectural approach to autonomic computing, *ICAC*, pp. 2–9.